A theory and model for the evolution of software services

Andrikopoulos, V.

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

A THEORY AND MODEL FOR THE EVOLUTION OF SOFTWARE SERVICES

A Theory and Model for the Evolution of Software Services

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit van Tilburg op gezag van de rector magnificus, prof. dr. Ph. Eijlander, in het openbaar te verdedigen ten overstaan van een door het college voor promoties aangewezen commissie in de aula van de Universiteit op vrijdag 1 oktober 2010 om 10.15 uur

door Vasilios Andrikopoulos

geboren op 16 januari 1981 te Patras, Griekenland.

Promotor: prof. dr. M. P. Papazoglou

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems (Dissertation Series No. 2010-45), and CentER, the Graduate School of the Faculty of Economics and Business Administration of Tilburg University.

This one goes out
to the ones I love[1]

[1]With my apologies to Michael Stipe and the rest of the gang for the misquote.

Contents

Contents i

List of Figures v

List of Tables vii

Listings ix

Preface xi

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Aim . . . 5
- 1.3 Scope . . . 6
- 1.4 Problem Definition & Assumptions . . . 7
- 1.5 Research Questions . . . 8
- 1.6 Research Methodology . . . 9
- 1.7 Contributions . . . 11
- 1.8 Structure of the Dissertation . . . 12

2 Background & Related Work 15
- 2.1 Software Evolution & Maintenance . . . 15
- 2.2 Software Configuration Management . . . 18
- 2.3 Evolution in Pre-Service Orientation Paradigms . . . 19
  - 2.3.1 Component-Based Systems . . . 20
  - 2.3.2 Object-Oriented Databases . . . 21
  - 2.3.3 Workflow & Process Management Systems . . . 22
- 2.4 Service Evolution & Adaptation . . . 23
  - 2.4.1 Corrective Approaches . . . 24
  - 2.4.2 Preventive Approaches . . . 26
- 2.5 Service Change Management . . . 29
- 2.6 Service Description . . . 30
  - 2.6.1 Web Services Description Languages . . . 31
  - 2.6.2 Other Initiatives . . . 32

2.7 Service Contracts . . . 33
2.8 Summary . . . 34

3 Running Scenario **37**
3.1 Description of the Scenario . . . 37
3.2 The Purchase Order Processing Service . . . 38
3.3 Evolutionary Scenarios . . . 42
3.3.1 Change Scenario I . . . 42
3.3.2 Change Scenario II . . . 43
3.3.3 Change Scenario III . . . 44

4 Service Representation **49**
4.1 Abstract Service Description Model . . . 50
4.1.1 Structural Layer . . . 50
4.1.2 Behavioral Layer . . . 52
4.1.3 Non-functional Layer . . . 53
4.1.4 Summary . . . 55
4.2 Formalization of the Abstract Service Description (ASD) . . . 55
4.2.1 Structural Layer . . . 55
4.2.2 Behavioral Layer . . . 59
4.2.3 Non-functional Layer . . . 60
4.2.4 Formal Definition of ASD . . . 61
4.2.5 ASD Consistency . . . 62
4.3 Discussion . . . 63
4.4 Summary . . . 63

5 Service Versioning **65**
5.1 Versioning in SCM . . . 66
5.2 Survey of Existing Approaches . . . 68
5.2.1 Version Identifiers and Version Space . . . 69
5.2.2 Versioning Methods . . . 71
5.2.3 Versioning Strategies . . . 73
5.2.4 Change Identification Model . . . 73
5.2.5 Findings . . . 74
5.3 The Versioned ASD Model . . . 75
5.3.1 Versioned Abstract Service Descriptions . . . 75
5.3.2 Representing the Version Deltas . . . 77
5.4 Summary . . . 78

6 Compatible Service Evolution **79**
6.1 Service Compatibility . . . 79
6.1.1 Introduction to Compatibility . . . 80
6.1.2 Formal Definition of Service Compatibility . . . 81

6.1.3 Supporting Techniques . . . 83
6.2 Type Theory for Abstract Service Descriptions . . . 85
6.2.1 A Short Introduction to Type Theory . . . 85
6.2.2 Structural Subtyping . . . 87
6.2.3 Behavioral Subtyping . . . 89
6.2.4 Non-functional Subtyping . . . 91
6.3 Reasoning on Service Evolution . . . 95
6.3.1 T-shaped Changes . . . 96
6.3.2 T-shaped Changes: Change Scenarios I-III . . . 98
6.4 Comparison with Existing Approaches . . . 101
6.4.1 Compatible Change Patterns . . . 101
6.4.2 Novelty . . . 103
6.4.3 Relevance . . . 104
6.5 Summary . . . 105

7 Service Contracts **107**
7.1 Service Contracts Life Cycle . . . 108
7.2 Interlude: A Consumer for the Purchase Order Processing Service . . . 109
7.2.1 ASD Representation of the Consumer . . . 111
7.2.2 Change Scenario IV . . . 112
7.3 Contract Formation . . . 113
7.3.1 ASD Views . . . 113
7.3.2 Matchmaking . . . 117
7.3.3 Contract Configuration . . . 119
7.3.4 Configuration Policies . . . 120
7.4 Service Evolution with Contracts . . . 121
7.4.1 Contractually-bound Evolution . . . 122
7.4.2 Contract Evolution . . . 124
7.5 Discussion . . . 126
7.6 Summary . . . 127

8 Validation **129**
8.1 Prototype . . . 130
8.1.1 Underlying Technologies . . . 130
8.1.2 Implementation . . . 131
8.1.3 Functionality . . . 131
8.2 Validation Experiment . . . 134
8.2.1 Setup . . . 135
8.2.2 Results & Analysis . . . 138
8.3 Realization . . . 141
8.4 Summary . . . 143

9 Conclusions & Future Work **145**
9.1 Summary 145
9.2 Research Results 147
9.3 Contributions 151
9.4 Evaluation & Limitations 154
9.5 Future Work 156

A Acronyms List **i**

Bibliography **iii**

Author Index **xix**

Index **xxv**

List of Figures

1.1 Structure of the Dissertation 13

3.1 Automotive Purchase Order Processing Scenario – BPMN Model (fragment) 39
3.2 Automotive Purchase Order Processing Scenario – UML Activity Diagram 40

4.1 The ASD Meta-model 51

6.1 Horizontal and Vertical Compatibility 81
6.2 QoS values relations 93

7.1 Contracts Life Cycle 108
7.2 ASD Views 113
7.3 Service Interaction 115
7.4 Contract Configuration 120
7.5 Contract Evolution – Backward Compatibility 125

8.1 SRM Meta-model in ecore format 132
8.2 SRM prototype – graphical editor 133
8.3 SRM prototype – reasoning module 134
8.4 PopService deployed in Axis2 service container 136

List of Tables

3.1 PopService Non-functional Properties (version 1.0) 42
3.2 PopService Non-functional Properties – Change Scenario I 43

4.1 ASD records summary 56

5.1 Approaches on service interface versioning 70

6.1 Guidelines for Backward Compatible Changes 84
6.2 Distribution of ASD elements $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ sets 97
6.3 Patterns of Change Sets 102

7.1 PopClient Non-functional Properties 111
7.2 Change Scenario IV – PopClient Non-functional Properties 112
7.3 Contract example between PopService & PopClient 122
7.4 Change Scenario I – using the Contract of Table 7.3 123
7.5 Change Scenario IV – Contract breaking 125
7.6 Change Scenario IV – Evolution of the Contract 125

8.1 Experimental validation results 138

Listings

3.3 PopService WSDL – Change Scenario I (`PODocument` only) 43
3.4 PopService WSDL – Change Scenario II 44
3.1 PopService WSDL file (version 1.0) 45
3.2 PopService BPEL file (version 1.0) 46
3.5 PopService BPEL – Change Scenario II 47
3.6 PopService WSDL fragment – Change Scenario III 48
4.1 POPService version 1.0 structural fragment 58
4.2 POPService version 1.0 behavioral fragment 60
5.1 Versioning examples of PopService in XML 72
5.2 Versioning example of PopService using UDDI tModel 72
6.1 Example of Schema Extensibility 84
6.2 PopService WSDL – Change Scenario I 88
6.3 PopService BPEL – Change Scenario II 90
7.1 PopClient Message Schema (version 1.0) 110
7.2 PopClient BPEL file (version 1.0) 110
8.1 Emfatic specification of the ASD Meta-model (fragment) 131
8.2 Alternative PopService Message Schema 139
8.3 Alternative PopService Message Schema – Change Scenario V 140

Preface

> Can't say I've ever been too fond of beginnings, myself. Messy little things. Give me a good ending any time. You know where you are with an ending.
>
> *A Kindly One, in Neil Gaiman's The Sandman*

Reaching the point of having this dissertation finalized and printed has been an interesting and eventful journey. As with all long-time endeavours, these last four years had their ups and downs. The course has been mostly steady though, and for this I have to thank a number of people that steered, helped, supported or simply existed around me. For these things (and many more) they deserve my thanks and acknowledgements.

First and foremost, I owe to my supervisor and promotor prof. Mike Papazoglou my deep gratitude and respect for not only giving me the opportunity to start a Ph.D., but most importantly for believing in me and supporting me in this effort. When I decided to start with the Ph.D. I used to say that he was the reason I joined the programme in the first place; I still stand fully behind this. Meeting and working with prof. Salima Benbernou has been a true blessing. It took us only a few hours to establish a common language and since then she's been – or at least that's how it feels to me – my unofficial co-supervisor, guiding me through the most esoteric parts of my work (and not only). Prof. Barbara Pernici has been very kind and helpful to me, not only for finding the time to work with me but also for being a member of my Ph.D. committee. My sincere thanks to prof. Willem-Jan van den Heuvel and to dr. Athman Bouguettaya for their comments and corrections on my dissertation – they have contributed significantly to bringing this book into shape and improving its quality greatly.

Through the S-Cube Network of Excellence I had the opportunity to meet and work with many interesting people (and a good portion of my committee). My special thanks go out to Martin Treiber in Technische Universität Wien and to Hossein Siadat, Mariagrazia Fugini and Pierluigi Plebani in Politecnico di Milano for sparing the time to work with me and collaborate on producing papers. Many thanks also to the Universität Stuttgart crew for their hospitality and useful take on my work.

Without the prompt help, contributions and Eclipse mastery of Juan Vara and David Granada in the Kybele Research Group at the University Rey Juan Carlos in Spain I wouldn't be able to finish the prototype discussed later in this dissertation in time. Thank you very much to both of you.

To my colleagues (and more importantly, friends) Amal Elgammal, Rafiq Haque, Michele Mancioppi, Khoa Nguyen, Michael Parkin, Oktay Türetken, Yehia Taher and Marcel Hiel, I owe many thanks for their help with reviewing the various chapters of this work and, together with former office mates Bart Orriëns and Benedikt Kratz, for all the interesting conversations, arguments and good times we had throughout the years. A very special acknowledgement goes to Alice Kloosterhuis, for resolving all issues at a glimpse and going beyond her duty to help me.

I feel I have been blessed to be surrounded by a number of wonderful people that have made my life much more pleasant. Andrea, Cristina, Willem, Renata, thank you for being my friends and for helping me survive the writing process with my sanity intact (more or less). Ákos and Aminah thanks, well, for adopting me (you're a regular surrogate family). Heejung and Marcel for bringing music into my life (best gift I ever got). Benjamin and Linde for all the booze (and not only). Maria G., Chris, Edwin, Gema, Olha, Dimka, Etienne, Alerk, Katie, Owen, Amar, Bianca, Jan-Willem, Jon and to the rest of the LG (associate and adjunct members included): thank your for the memories! A big shout-out to the Greek community abroad – Maria K., Giorgo, Sotiri – and in the old country – Sofia, Natassa, Vicky, Giorgo, Vaso and Mimi.

Last but not least, my deepest gratitude goes out to my family: my parents Niko and Anastasia and my siblings Dimitri and Katerina. Words are simply not enough to thank them for their love, support and faith in me. I just hope one day I will be able to somehow repay them.

On a closing note: everyone that has spent more than a few hours working with me finds out sooner or later that I can't function properly without some music in the background. Music influences and interacts with the way I think. I suspect that there are whole paragraphs in this document that, at least originally, are following the rhythm of a particular song. This work is therefore incomplete without its soundtrack which contains lots of Red Hot Chili Peppers, early Queen and post-Blackwater Park Opeth, many repeats of The Ocean's Heliocentric, Rush's Snakes and Arrows, TV on the Radio's Dear Science and Queens of the Stone Age's Rated R, topped with assorted listenings to various records of The Police, The Kilimanjaro Darkjazz Ensemble and Dream Theater (particularly their brilliant covers). Feel free to play them along as you read it.

Vasilios Andrikopoulos, September 5, 2010.

Chapter 1

Introduction

> For an evolutionary system, continuing development is needed just in order to maintain its fitness relative to the systems it is co-evolving with.
>
> *The Red Queen Hypothesis*

> Time condemns us to change. We would rather not change, but we have no choice.
>
> *Balthasar Holz*

Change has been one of the major themes in the evolution of human civilization. Different cultures establish different mechanisms for coping with the perceived struggle between Status – the established order of things – and Change – the chaotic, and without necessarily a clear destination at times, movement. Western civilizations, influenced by the highly organized and compartmentalized thinking of certain philosophical schools, tend in principle to perceive change as a transitional period between states. Eastern cultures on the other hand perceive change as a natural part of the flow of life and, in some ways, as more important than the in-between states themselves.

This struggle between established order and chaotic movement brought about by change did not take long to manifest in software engineering. Even as the first large scale software projects were being developed for the early mainframes, the limited time span of the results and the volatility of system design quickly became apparent. To quote Frederick Brooks Jr. [1]:

> Once one recognizes that [...] a redesign with changed ideas is inevitable, it becomes useful to face the whole phenomenon of change. The first step is to accept the fact of change as a way of life, rather than an untoward and annoying exception.

"Classic" engineering disciplines have little room for change. From the moment a design is made and the construction of the project begins, changes to project blueprints are to be avoided at all costs. In case such a change is deemed absolutely necessary (in cost-benefit analysis terms), then the construction halts and a redesign takes place. Since design takes in principle much less effort than construction, this process can be repeated a number of times in the span of a large project – assuming that it does not translate into large increases in the construction time and cost.

Attempting to apply a similar approach in software development created – and still creates – a lot of confusion and disappointment. This is not very surprising if one considers the reversed relationship between design and construction time and cost in the software domain. While in traditional engineering the construction time and cost is proportionally dominating the one for design, in software engineering this relationship is inversed [2]. Stopping construction and returning back to the design phase may cause an increase to the total time and cost by orders of magnitude. Software engineers have realized this problem quite early on. Applying rigid techniques in order to limit the exposure to change in the design of large systems has been proven limited. As pointed out by David Parnas [3]:

> Our ability to design for change depends on our ability to predict the future. We can do so only approximately and imperfectly.

As a result, a new strategy emerged in software engineering: instead of attempting to contain change on the system design level, better decompose the system into smaller units that are affected to a less dramatic extent by changes.

Change in Service-Oriented Systems

This evolutionary trend calling for more decentralized systems that cope with change in smaller scale through encapsulation and reusability, resulted in the paradigm of service orientation [4]. The purpose of Service-Oriented Architecture (SOA) is to address the requirements of loosely-coupled, standards-based and protocol-independent distributed computing, mapping enterprise information systems appropriately to the overall business process flow [5]. Of course, as with all other efforts following the same trend (e.g. distributed computing, object orientation, component-based systems), the SOA paradigm comes with its own set of problems. Decomposing the system into smaller units scales down the effort of developing and coping with changes to a local, per unit basis. This decomposition however comes with a trade-off which is expressed as an increased effort in developing and maintaining the interconnections between units. The working assumption in this case is that the aggregate effort across all decomposition units is less than the effort required by a monolithic system to deal with the same situation.

The similarity of a service to an individual organism that depends on its environment to perform certain functions invites the comparison between the evolution of services and the evolution of organisms. While this metaphor can not be taken very far (due to the critical differences between individual organisms and software artifacts – like the ability

to procreate and transfer its characteristics to its successors [6]) the idea of dealing with change as part of a natural process in the life time of services (or any other software artifact for that matter) appears sound. *The contribution of this dissertation is to define what constitutes evolution in software services, and more importantly, how can this evolution be constrained and kept consistent when services transit from one version to another in piecemeal fashion.*

The rest of this chapter is organized as follows: in Section 1.1 we motivate this research by showing that SOA is intrinsically connected to change and by discussing the innovation of our work. In Section 1.2 we define the goals of our research and in Section 1.3 we define its boundaries. Section 1.4 provides a definition of our research problem based on the context discussed in the previous sections, and the assumptions we make in dealing with it. The methodological approach taken is discussed in Sections 1.5 and 1.6, where the problem is decomposed into discrete research questions and the methodology for addressing them is presented. The contributions of this work are summarized briefly in Section 1.7, before closing the chapter with the outline of the dissertation in Section 1.8.

1.1 Motivation

Software services are subject to constant change and variation. By implementing and automatizing business processes, services are subject to continuous adaptation in order to deal with the serious challenges of the enterprise environment. Mergers and acquisitions, outsourcing possibilities, rapid growth, regulatory compliance and intense competitive pressures are overtaxing traditional business processes and hinder innovation and alignment with the enterprise goals and strategies [7].

With respect to these challenges, SOA is being perceived as an enabler of business flexibility. Mergers and acquisitions for example are facilitated on two levels. On a tactical level, SOA-based approaches integrate the core functionality through the use of eXtensible Markup Language (XML) schemas to normalize and exchange information between parties. On a strategic level, they adopt standardized services, with custom services developed only for specialized requirements. This is a more efficient approach than selecting the best of breed applications and standardizing them where and when possible [8].

SOA increases an organization's agility by encapsulating business functions in well-defined, reusable and visible across the organization services. These services are then connected (composed) in order to implement core business processes. The cost of change is decreased by minimizing the dependencies between services and allowing them to be recomposed on demand. An organization can only fully realize these benefits, however, if its SOA instantiation enables services to evolve independently of one another [9]. To accommodate the volatility of the business environment it is required of services to be able to continuously evolve and respond to environmental demands without compromising operational and financial efficiencies [10].

In addition to the external demand for change, services have also to deal with internal evolutionary pressures. Services for example are software artifacts and as such they

age: they grow bigger and more complicated, and as a result, their consumers experience reduced performance and reliability [3]. Reacting to this process and stopping the deterioration is as necessary as proactively accommodating change by design. Furthermore, as producers and consumers of a language, services are subject to the evolution of the language itself. The W3C Technical Architecture Group (TAG)[1] under David Orchard [11] attributes this evolution to fixing bugs and other errata, dealing with changing requirements, providing desirable variations of the language and performing readjustments to fit the implementation. Whatever the reasons, different versions of the language – and therefore of the service providers and consumers that communicate in this language – will appear over time. An appropriate evolutionary strategy is therefore required to deal with them.

Dealing with change in a service-oriented environment includes many different issues, from purely technical (in terms for example of message schemas to be exchanged) to organizational (e.g. managing the transitional period between different service releases in an enterprise). As noted in [12], the services life cycle itself is characterized by its highly dynamic features: new services are created without any notification, providers interrupt the provisioning of services without any indication and the functionality of services changes overtime. Changes can happen at any stage in the service life cycle and have an unpredictable impact on the service stakeholders. Being therefore able to control how changes manifest in the service life cycle is essential for both service providers and service consumers.

Furthermore, the distinct roles and the independence of service providers and consumers in the SOA paradigm means that each entity must be considered separately when a service is to be updated. When changes affect the service provider's application system, service consumers typically do not perceive the upgrade of the service immediately. The change is usually later identified by its effect on the consuming applications. Consequently, Service-Based Applications (SBAs) consuming an upgraded service may fail on the service client side due to changes carried out during the service upgrade. In order therefore to manage changes in a meaningful and effective manner, the service consumers using a service that needs to be upgraded must also be considered when service changes are introduced at the service provider's side. Failure to do so will most certainly result in severe application disruption.

Unfortunately however, the management of change in the context of service orientation has not been discussed sufficiently so far. As we will discuss in Chapter 2, existing works approach service change from a classic software engineering perspective trying either to describe how services can adapt to accommodate change or to prescribe what type of changes are allowed to a service to avoid disruptions. All of these works are based on empirical findings and best practices to deal with change, usually relying on the specifics of the technologies used to achieve their purposes. This modus operandi leaves these approaches vulnerable to technological shifts and without a theoretical foundation that transcends the minutiae of each technology they rely on. This is a need that this work is

[1] `http://www.w3.org/2001/tag/`

geared to address. In the following sections we discuss more specifically how we approach the management of change in services.

1.2 Aim

As described above, dealing with change in an SOA environment includes different technical and organizational issues. For this reason we focus our interest in managing *service evolution*, defined in [7] as *the continuous process of development of a service through a series of consistent and unambiguous changes*. Service evolution is expressed through the creation, provisioning and decommissioning of different variations of the service called *versions* during its life time. These versions must be aligned with each other in such a way as to allow a service designer to track the various modifications that have been introduced over time and their effects on the original service. To control service development, a developer needs to know why a change was made, what its implications are, and whether the change is complete.

An approach to managing service evolution must therefore rely on a framework that controls and manages service changes in a uniform and consistent manner. This ensures new and old service versions can co-exist, are reliable and well-behaved, and will not disrupt clients that are using them. Service evolution management thus requires an understanding of all the points of change impact, controlling service changes, tracking service versions and reasoning about their status [13]. Service evolution management entails continuous service re-design, re-engineering and improvement effort. This effort however should not be disruptive for the consumers of the service and should not interfere in the way that SBAs that use the upgraded service perform. Ideally, no radical modifications must be required in the very fabric of existing services to accommodate change. Nevertheless, even routine service changes, such as the introduction of new functionality, increase the propensity for error.

Eliminating spurious results and inconsistencies that may occur due to uncontrolled changes is thus a necessary condition for the ability of services to evolve gracefully, ensure service stability, and handle variability in their behavior. With the above in mind, we can classify service changes depending on their causal effects as [7]:

1. *Shallow changes*: Small-scale, incremental changes that are localized to a service and/or are restricted to the consumers of that service.

2. *Deep changes*: Large-scale, transformational changes cascading beyond the consumers of a service possibly to consumers of an entire end-to-end service chain.

Ensuring a change is shallow requires service developers to reason about the effect of the change on the service consumers. The number, type and specific needs of the consumers is often unknown and their dependencies on the service are transparent to the developer. This reasoning can only be performed on the basis of a set of formal principles that define what constitutes shallow change. A formal approach for shallow changes is currently unavailable and this work aims to address this need. The goal of this research is therefore:

> *to provide a theoretical framework for service developers so that they can develop evolving services that constrain the effect of changes to a service so that it does not lead to inconsistent and spurious results and does not disrupt its service clients. In particular we shall constrain our work to addressing the effects of shallow changes.*

Deep changes on the other hand are quite intricate and according to [7], they require the assistance of a change-oriented service life cycle to allow services to respond appropriately to changes as they occur. This life cycle is concerned with analyzing the effects and dealing with the ramifications of operational changes and changing compliance requirements which rely on service composition re-engineering exercises. Due to the complexity and magnitude of the issues involved in the management of deep changes this work is limited to shallow changes.

1.3 Scope

An integral part of the SOA paradigm is the principle of encapsulation which dictates the separation of concern between the *description* and the *implementation* of the service. The former is concerned with the contractual interfaces that the service is exposing to its clients, usually defined in document(s) of one or more of the widely accepted set of standards, like Web Services Description Language (WSDL). The latter focuses on how services are implemented, either as atomic or composite services. A variety of options exist for the technologies and architectural styles to be used in implementing an atomic service. Most of them are however firmly grounded in vendor-promoted solutions like the Java J2EE or the Microsoft .NET environments. On the other hand, composite services are often implemented by a combination of executable Business Process Execution Language (BPEL) service compositions, software components and service container-specific "gluing" scripts.

Due to the encapsulation of services, implementation changes are transparent to the service consumers if they do not affect its description. Changes to the service implementation are therefore of concern only when they have an impact on the service interfaces. Additionally, despite the fact that standards like BPEL describe how service compositions can be implemented, there are no widely-accepted standards governing service implementation in general. Discussing therefore the evolution of service implementation would confine us to specific technological solutions and detract from the generality of our approach. For these reasons we scope our investigation on service evolution exclusively to service description.

Services typically evolve by accommodating a multitude of changes along the following, non-mutually exclusive dimensions:

1. *Structural changes* focus on changes that occur on the service data types, messages and operations, collectively known as the service signatures.

2. *Behavioral changes* affect the business protocol of a service. Business protocols specify the external messaging and perceived behavior of services (viz. the rules that

govern the service interaction between service providers and consumers) and, in particular, the conversations that the services can participate in.

3. *Policy-induced changes* describe changes in policy assertions and constraints on the invocation of the service. The offered Quality of Service (QoS) characteristics of a service, for example, are expressed in terms of policy assertions. Policies may also describe constraints external to those agreed by the interacting parties. These constraints may include universal legal requirements, sectorial requirements and contract terms, public policies (e.g., privacy/data protection, product or service labeling, consumer protection) and laws and regulations that are applicable to parts or the whole of a service.

4. *Operational changes* concentrate on the spreading effects of changing the nature of service operations. For a change in an order processing operation, for example, this requires, among other things, the understanding of where time is consumed in the manufacturing process, what is "normal" with respect to an event's timeliness as regards the deadline, and understanding standard deviations with respect to that manufacturing process' events and in-time performance.

Structural, behavioral and QoS-related policy-induced changes refer to the externally observable aspects of a service (in terms of its signatures, protocols, etc.). These types of changes have a direct and profound impact on the service interfaces and as such they will be discussed extensively in the following chapters. Changes due to legislative, regulatory or operational requirements on the other hand require a deeper understanding of the inner workings of the service and the organization that provides it and for this reason they are outside of the scope of this work.

1.4 Problem Definition & Assumptions

From the previous discussion it emerges that there is a clear necessity for ensuring that the changes which occur while a service is evolving are shallow. A mechanism is required for analyzing the proposed (or already applied) changes to a service and concluding whether they are shallow or not. For this purpose we use the principle of *service compatibility*. The fundamental assumption is that a change is shallow as long as it results in a service that respects a set of predefined compatibility criteria:

> **Hypothesis:** Changes that preserve the compatibility of services (for a given definition of service compatibility) are shallow.

Given the diverse use and overloading of the term "compatibility" in the literature we will refrain at this point from exhaustively defining what compatibility entails. In this section we will use for convenience the colloquial definition of compatibility as "the capability of orderly and efficient integration and operation with other elements in a system with no

required modification or conversion"[2]. In the following chapters we will perform an investigation into different aspects of service compatibility. The purpose of this investigation is to provide a formal definition and to identify the theoretical underpinnings of the term as the means to achieve the goal of this work, namely:

> **Problem Definition:** Under which conditions can services evolve while preserving compatibility?

In order to properly define the boundaries of the problem we are making the following assumptions:

1. Services are treated as black boxes with respect to their composition and/or their implementation. Any change occurring to these aspects is important for this discussion only as far as it has an effect on the externally perceived aspects of the service.

2. Everything of concern to this work can be described as a service. SBAs, resources, etc. expose the same type of interfaces as software services. This allows us to deal with them in a uniform manner.

3. Commonly agreed semantics have been established for the message payload in the interactions between service providers and consumers. This means that either the semantics of the service signatures follow an accepted set of conventions, or that they refer to the same knowledge model, or both. Matching heterogeneous semantics in the description of a service is outside the scope of this work.

4. Due to the encapsulation and loosely coupled properties of SOA, no information is available externally about the operational semantics of the service, i.e. what computational steps are taken by the service. Only the perceived behavior of a service in terms of its interactions with its environment (clients and other services in the service chain) are considered.

1.5 Research Questions

In order to provide a solution to the problem of service evolution as defined above we decompose it into the following research questions:

1. *What is the State of the Art in service evolution and how is evolution treated in relevant research fields?* What are the techniques, theories and lessons that can be taken from the literature and the industrial practice?

2. *How can evolving services be represented in a uniform manner?* What are the dominant trends in service interface description and how do they incorporate service evolution?

[2]The American Heritage Dictionary of the English language, Fourth Edition

3. *What exactly constitutes service compatibility?* A theoretical and practical definition of compatibility in the context of services is required to allow the definition of when evolving services are compatible.

4. *What are the conditions that enable compatible service evolution?* How does the definition of service compatibility interact with the evolution of services? How is it possible to constrain the type of changes to a service to a set of compatibility-preserving ones? What are the benefits of this evolutionary model with respect to the State of the Art?

5. *Is service compatibility equivalent to shallow changes? Are there alternative models of shallow changes outside of the service compatibility one?* Can the restrictions to the allowed changes to a service be relaxed? What are the benefits and disadvantages of such a solution?

6. *How can the proposed solution be validated practically? Can the theoretical results be replicated by a prototype? What are the limitations of the proposed solution?* A proof-of-concept implementation is required in order to demonstrate the realization of the solution. Furthermore, an evaluation of its realization with respect to existing technologies and standards is necessary.

1.6 Research Methodology

This section outlines the methodology used to conduct this research. Conceptually, providing service developers with the means to control the evolution of services belongs to design science [14]. In the nomenclature of design science, we aim to construct a method that allows for the selection of compatible service versions. This method can be instantiated into a prototype that provides a "proof by construction" of the feasibility of the designed method. As we discussed however during the definition of the problem, the approach we use depends on the definition of a theory for the compatibility of services. Theories are traditionally outside the scope of design science [15] and they put emphasis on rigor at the expense of relevance [14]. Since we aim to combine theoretical with engineering methodologies for developing our solution we also combine different steps in our methodological approach.

In particular, we opted to decompose our research approach into five distinct steps:

Step 1: Problem Definition

In any research effort, the first step is to understand and properly define the problem at hand. The short-term goal is to obtain a clear picture of the domain of the problem and formulate a preliminary hypothesis that will allow further investigation. As research progresses, both the definition and the hypothesis will evolve into more concrete forms. The problem definition and the research questions discussed in this chapter are the culmination of this effort.

Step 2: Establishment of State of the Art

The investigation and analysis of existing literature on the defined problem domain and on related research fields serves two purposes. First, it helps to better establish the scope of the research that leads to further refinement of the problem definition. Second, it allows the identification of both best practices and open issues through the categorization of existing solutions that enables the grounding of the research in other efforts. To establish the State of the Art, both (academic) publications and industrial efforts are considered and presented in Chapter 2.

Step 3: Solution Design

Having established in the previous step the scope of the research and the strengths and shortcomings of existing solutions, this step calls for the design of a solution to the problem. This involves developing a model for the evolution of services supported by a theory for service compatibility, as discussed in Chapters 4, 5 and 6. The theory developed is then applied to an alternative model for the interaction and evolution of services that allows for additional flexibility in Chapter 7.

Step 4: Validation

The validation of the solutions proposed in this work is performed at three levels. On the first level, the formal underpinnings of the proposed solutions (it terms of the developed model and theory) ensure its logical consistency throughout Chapters 4 to 7. On the second level, a running scenario taken from a complex business case is presented in Chapter 3 and used throughout this work to demonstrate the usability of the approach. Finally, on the third level, the realization of the solution is demonstrated in Chapter 8 through a proof-of-concept prototype that illustrates the changes required by the dominant service description standards in order to realize the full potential of our solution.

Step 5: Evaluation

As a last step of this work, an evaluation of the proposed solution in terms of its benefits and shortcomings is performed in Chapter 9. During this step we identify where this approach fits in the State of the Art, and in what ways the solutions proposed progress it. Future directions are also identified in connection with not only the problem solution but also to possible applications of the research results to other problem domains.

These steps are not sequential tasks but rather iterative in nature, requiring revisit and refinement as research progresses and new information is collected.

1.7 Contributions

The results of this work address the need for a comprehensive, theoretically-supported model for the management of service evolution. A set of theories and models that unify different aspects of services into a common reference framework for the representation, versioning and evolution of services has been developed for this purpose. This framework pushes forward and redefines the State of the Art in service evolution. It achieves this by replicating and formally validating the empirical findings and best practices for service evolution. At the same time it outlines a number of possibilities for service evolution that are not currently covered by existing standards and technologies.

A preliminary list of the contributions of this research is presented here. This list will be further discussed in the closing chapter of this book. The major results of this work with respect to the State of the Art in service evolution and service science are:

A technology-agnostic uniform formal model for the representation of service interfaces and their different versions. The service representation model developed seamlessly integrates the different aspects of services (structural, behavioral and non-functional) into one model. The model is augmented with the means for versioning a service at different granularity levels. The survey on *service versioning* can also be considered as an important contribution to this field.

A theory and model for the compatible evolution of services. The major contribution of this work is the identification and formalization of the conditions under which services can evolve while preserving their compatibility. The conditions are expressed as permitted sets of changes that can occur safely to a service. Both an informal and formal definition of service compatibility is provided based on a combination of type and set theory. The theory developed is a sufficient condition for ensuring the shallow nature of changes.

A contract-based model of service interaction and evolution. Service contracts are introduced between service providers and consumers as bilateral agreements that specify explicitly the expectations and obligations of both parties. Based on these contracts an alternative evolution model is proposed that expands the permitted set of changes and provides more flexibility in evolution – at the trade-off of increased coupling, governance and communication overhead.

An identification of the limitations of existing specifications and technologies with respect to service evolution, and a proposal for their improvement. The dominant language specifications for service description were evaluated on the basis of their support of compatible service evolution using the findings of this research as a benchmark. As a result, a proposal for their improvement is put forward.

1.8 Structure of the Dissertation

The structure of this work is summarized in Fig. 1.1. Chapter 1 introduces, discusses the motivation and defines the problem and the methodology used to develop the solution. Chapter 2 sets the background by examining related efforts in the evolution of services and software in general. The chapter that follows (Chapter 3) presents a running scenario based on an industrial case study that will be referred to a number of times in the following chapters. Chapters 4 and 5 discuss how to represent a service and its evolutionary history in terms of its versions, respectively. Using the models developed in those chapters, Chapter 6 defines service compatibility and develops a theory for the compatible evolution of services.

Chapter 7 presents an alternative model for managing the evolution of services using bilateral agreements between service providers and consumers. Chapter 8 evaluates the feasibility of the approach by presenting a proof-of-concept implementation, and discusses its realization with respect to existing technologies and standards. Finally, Chapter 9 concludes by summarizing the findings, assessing them against the research questions posed in the introduction and by briefly discussing future research directions.

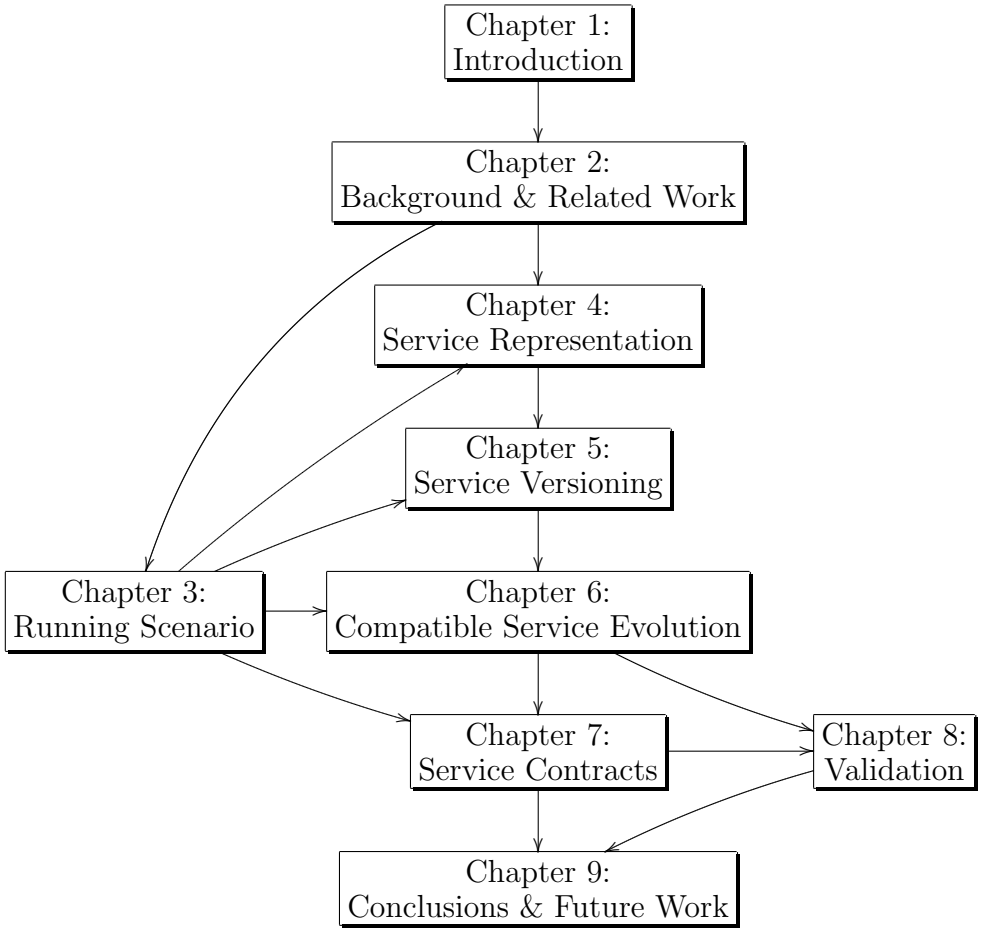


Figure 1.1: Structure of the Dissertation

Chapter 2

Background & Related Work

> Those who would repeat the past must control the teaching of history.
>
> *Bene Gesserit Coda*

> The tendency in evolution is toward greater and greater specialization. [...] Too much knowledge has piled up in each field. And there are too many fields.
>
> *Philip K. Dick*

In the following we establish the background of our work and we discuss related works in service evolution and relevant fields. The chapter starts by investigating and discussing evolution in software, component-based, object-oriented and workflow management systems. Then we survey the existing approaches on service evolution in order to set the background for this work and clarify its scope. For this purpose we also discusses different takes on (service) change management. While not necessarily in the scope of this work, these works offer valuable lessons and techniques for constructing a proper compatible service evolution solution. Moving on, we present the related work on the description of services, which we need in order to discuss the respresentation, versioning and compatibility of services in the chapters that follow. We also introduce the notion of service contracts that we will come back to in Chapter 7. The chapter closes with a brief summary of the main points that were presented.

2.1 Software Evolution & Maintenance

Evolution in software systems has been traditionally considered as either a part or a synonym of software maintenance [16]. Starting from this assumption, works on software evolution like for example [17] and [18] expanded the classical work of Swanson et al. [19] and [20] to build taxonomies of change. Their goal is to analyze the different aspects of iterative change to software and diagnose/predict the factors that govern it.

In [17] for example, a decision tree is presented that summarizes the classification of different types of software evolution and maintenance into clusters (support interface, documentation, software properties and business rules). Each cluster groups the decisions to be made and associates a type of evolution/maintenance to each of them. Answering positively to the question "Did the activities use the software as a basis for consultation?" for example characterizes the change as consultive. Change types inside the clusters are prioritized based on their impact, and the effort required for each change can be estimated by navigating the tree. The authors recognize evolution and maintenance as separate activities but they prefer to treat them as one in their categorization. In this work we focus on the evolutionary process rather than the reactive/proactive diagnosis and intervention approach of maintenance.

The term *evolution*, as reported in [17], had already appeared in the 1960s to characterize the growth dynamics of software. It was popularized by Belady and Lehman when discussing their empirical experiments on the IBM OS/360 system using a series of releases. The insight gained by these studies is that software evolution could be systematically studied and exploited. This also resulted in three originally, and later extended to eight, laws [21] that drive and govern the evolution of software systems:

1. *Continuing change:* systems must be continually changed, otherwise they become less satisfactory to their users.

2. *Increasing complexity:* the evolution of a system leads to more complexity – except if some sort of maintenance procedure is applied to it.

3. *Self regulation:* the evolutionary process is regulated by the system itself.

4. *Conservation of organizational stability:* the average effective global activity rate in a system is invariant during a product's life time.

5. *Conservation of familiarity:* the average content of successive releases of a system is invariant during its life time.

6. *Continuing growth:* the offered functionality of a system keeps increasing.

7. *Declining quality:* the perception of the system if it evolves uncontrollably is of lower and lower quality.

8. *Feedback system:* the evolution processes are multi-level, multi-loop, multi-agent feedback systems.

The reader is referred to [22] for a recent discussion on the history and evolution of the theory. What is important for this discussion is that evolution is treated as a process of continuous change applied to a system that is in a feedback loop with the system itself [21].

As concluded by Mittermeir [23], systems evolution is driven by two forces: *market factors* (or other comparable social phenomena), and *technical factors* (that are essentially also human-controlled). Market factors are not confided to the financial aspect of the system environment, but they may also include legislative and social changes. The prematurity of organizations can also act as a negative factor in evolution, expressed as a disability or reluctance to accept new types of systems. The technical factors of system evolution are driven by the inability of the system stakeholders to control the extent to which technological changes become commonly acceptable. This categorization essentially confirms the necessity for evolution that we discussed in the introductory chapter.

Evolution is particularly important in distributed systems due to a complex web of software interdependencies. As Bennet and Rajlich point out [24], attempting to apply the conventional maintenance procedure (halt operation, edit source and re-execute) in large distributed systems (like the ones emerging in service-oriented environments) is not sensible. On the one hand, the difficulty of identifying which software artifacts form the system itself is non-trivial, especially in the context of large service networks. In addition, the matter of ownership and access to the actual source code (if any) of third-party services that is directly linked to the encapsulation and loose coupledness promoted by service orientation does not easily allow the application of many of the maintenance techniques like refactoring [25] or impact analysis [26], [27]. Towards that direction Bennet and Rajlich [24] decompose maintenance into *evolution* and *servicing* and treat the former as an iterative development phase and the latter as the more traditional post-development corrective, perfective and preventive actions. This distinction is respected in the context of this work.

Drawing inspiration from the field of biology where evolution is one of the core concepts, a number of works like [28] and [6] attempt to draw analogies with different branches of biology and apply methods and techniques from it to software. Functional paleontology [28] in particular studies the telephony services domain for a period of 50 years as a fossil record of sorts. They analyze the evolutionary patterns that emerge from this record showing the interplay between different types of features and provide evidence for the punctuated evolution of the services domain (i.e. that of abrupt expansive phases followed by periods of relative stability). More importantly, they demonstrate that different change drivers operate at different speeds, resulting in unbalanced and spurious development of certain features at the expense of others.

On the other hand [6] compares directly the biological (as perceived by the Darwinian perspective and its descendants) and software evolution for similarities and discrepancies. They conclude that despite the fact that many aspects of the natural kingdom are exhibited as part of software systems, the inability of software artifacts to be identified as coherent individuals limits the application of biological theories to software evolution – at least in its traditional form. The individualization of software into components and services, and the relevance of the telecommunication industry and its pioneering work on services make both these works quite important contributors to the discussion of service evolution.

2.2 Software Configuration Management

Software Configuration Management (SCM) is the discipline of controlling the evolution of complex software systems [29]. SCM has contributed in major ways to software maintenance and evolution [24] and for that reason it has to be taken into consideration when discussing service evolution. The term SCM denotes the discipline of control of the evolution of complex (software) systems that since the late 90s focuses on supporting *programming in the wide* [30]. Instead of focusing though on the analytical and predictive aspect of the management of software evolution as the approaches in the section on Software Evolution did, the emphasis here is on the coordination and support of development.

SCM systems were originally used for managing critical software, usually by a single person on one mainframe computer. A custom system was usually developed as a result of the need for supporting the building of different versions of the software. With the advent of distributed computing and the popularity of operating systems like UNIX, the focus changed in supporting large-scale development and maintenance by groups of users, which created in turn the need for workspace management. This need was again served by mostly ad hoc solutions. Currently, SCM systems are responsible for managing the evolution of any kind of software, developed on any number of machines by a number of users that are probably in distributed locations [31]. A number of tools and systems integrating explicit process control have been developed for this purpose that the authors of [32] and more recently [31] survey exhaustively.

In the domain of SCM, the approach proposed in this work shares a number of conceptual similarities with the NuMIL language by Narayanaswamy and Scacchi [33]. NuMIL is combining SCM and software evolution techniques to deal with evolving software systems by maintaining the integrity of their configurations while they evolve. Integrity in this context is essentially equivalent to compatibility between configurations. In order to properly define and reason on compatibility, they abstract from the particular system description language and they use a theoretical model for describing the interfaces of the (sub)systems. Based on these abstract descriptions and using a set of defined formal properties, they are able to decide whether a new system configuration is compatible with the previous ones or not. They also use the notion of upward compatibility as the means for controlling the incremental development of software systems. Our notion of compatible service evolution and the reasoning on it based on abstract service descriptions can be seen as the evolution of these ideas for SOA.

Furthermore, given that services are becoming more complex to compensate for increasing business needs, valuable lessons and techniques can be drawn from SCM for service evolution management. From the aspects that have been developed under the SCM umbrella, of particular interest for the service evolution is the product support in terms of *versioning* as summarized in [31]. More specifically, versioning refers to the keeping of a historical record of the software artifacts as they undergo change and is the fundamental block of SCM. The reliance of SOA on the publishing of service interface descriptions (e.g. in WSDL) and interaction protocols (in Abstract BPEL) – as we will discuss in later sections, together with the predominant use of XML as the description language, adds an

additional dimension to the versioning of services . In particular, it requires the promotion of structured documents to first-class software objects that need to be versioned and related to the other objects (e.g. documentation, code, test-related documents). These documents are the only means of interaction with the service and confine for that purpose the executable code to an internal to the service role.

Traditional SCM systems fall short in supporting distributed environments like services in two particular aspects [34]:

1. they tend to focus on the file artifact as a first-class citizen and ignore higher levels of abstraction and organization, and

2. they assume a centralized control with respect to the evolution of the software artifacts.

These shortcomings are serious obstacles in applying SCM technology "as-is" to services. While services are usually described in terms of documents, the description of the service may span multiple documents. Furthermore, sections of each document may be re-used for the description of a number of different services across multiple organizations. The actual content of each document is as such more important than the document itself. Even more critically, the services that are used for the implementation of the service can be provided by other departments within the organization or by other organizations altogether. Control of the development and provision of these consumed services is in principle out of the hands of one service developer and distributed across the different organizational units. A distributed, content-centric approach is therefore required in supporting versioning in SOA.

Nevertheless the methods and techniques developed for versioning in the field have been proven irreplaceable and the tools for supporting it are pervasive. Contemporary revision control systems like the popular CVS[1] and Subversion[2], or their modern distributed counterparts like GIT[3], Mercurial[4] and Bazaar[5] (among others) are indispensable for collaborative development of software. As such, they are very useful for controlling the evolution of the service implementation but as discussed in Chapter 5 they are very limited in supporting the versioning of service interfaces.

2.3 Evolution in Pre-Service Orientation Paradigms

This section is briefly discussing research fields that are tightly related to software and service evolution for different reasons. The purpose of this discussion is to identify aspects of each field that are useful for service evolution.

[1] `http://www.nongnu.org/cvs/`
[2] `http://subversion.apache.org/`
[3] `http://git-scm.com/`
[4] `http://mercurial.selenic.com/`
[5] `http://bazaar-vcs.org/`

2.3.1 Component-Based Systems

The difficulty of approaching the evolution of systems as a purely maintenance activity has already appeared in the study of component engineering in general, and component evolution in particular. For the term (software) component we use the widely accepted definition of [35] as "binary units of independent production, acquisition and deployment that interact to form a functioning system". Components can therefore be perceived as coarse-grained opaque software artifacts that contractually specify their provided and required interfaces. OMG's CORBA, Sun's JavaBeans and Enterprise JavaBeans and Microsoft's COM and DCOM (subsumed by the .NET framework) infrastructure technologies have matured enough to become standardized [36]. Component-Based Systems (CBS) are driven by the idea of industrializing the software development process by transforming it into an assembly of existing parts. CBS are in principle geared towards dealing with change by depending on the quick assembly of applications out of prefabricated components and the availability of large collections of interoperable software components [37].

The survey of [38] summarizes the basic ideas and solutions for the evolution of CBS. Evolving a component for example includes changes in both its interfaces and its implementation, with each one of these aspects having different evolutionary requirements. Due to their composability and emphasis on reuse, components exhibit strong dependencies with the other components that they consume. Changing a component may therefore have implications to other components, and upgrading to a new component may require for both versions (old and new) to be deployed in parallel while the transition takes place. Finally, identifying and distinguishing between different versions of components require the introduction of SCM techniques, like version identifiers incorporated into e.g. the component meta-data. Since version identifiers do not explain what changes occurred between versions, checking for compatibility has to be performed separately.

Historically and conceptually, CBS can be considered as a predecessor of SOA, which in turn expands and builds on the same principles of encapsulation, independence and unambiguous definition of interfaces. However it has to be kept in mind that components and services are quite different in terms of coupling, binding, granularity, delivery and communication mechanisms and overall architecture [37], [39]. The applicability of a component evolution theory or technique as summarized by [38] for example should always be examined carefully before adopted.

Investigations into component evolution have warned about potential pitfalls in the proliferation of a distributed environment like CBS and SOA. As reported in [40] for example, the use of components may provide short-term effectiveness but introduce long-term problems in reusability and maintainability - exactly the issues that they were meant to solve. In addition, the cost to maintain CBS (and in particular Commercial-Off-The-Shelf (COTS) systems) equals or exceeds that of developing custom software and maintenance complexity (and costs) increases exponentially as the number of components in the system increases [41]. For these reasons [16] concludes that evolution in component and service oriented systems should shift its focus from the code-changing perspective to that of the artifact-replacement one. Our approach is facilitating this transition by providing the means for

analyzing, evaluating and constraining the impact of such a replacement in terms of the service interfaces.

Of particular interest and relevance to our approach are the works of [42], [43] and [44] that discuss the evolution of components based on types. Types in this context are sets of components that exhibit the same behavior. A (component) type system constrains the component structure and behavior in order to guarantee the logical consistency of the components. These works build a type system for components and on top of this system they define a set of conditions under which components can evolve in a compatible way. As with the other works on CBS however, their solutions can be only loosely applied to SOA. The fine granularity required for a generic component type system and the emphasis on preserving the operational semantics of the component during the evolution – at the expense of the perceived behavior and interface signatures – are major obstacles for adopting these theories unchanged in the scope of this work. In this sense they can only be considered as conceptual predecessors of our approach.

2.3.2 Object-Oriented Databases

Due to their historical position and influence in modern system thinking, Object-Oriented (O/O) systems are essentially the progenitors for both component- and service-orientation. As such, research in O/O systems had to deal with many issues that later re-appeared in different forms. O/O databases in particular, having to deal with persistent, long-lived objects that were by necessity forced to evolve over time, have developed a number of solutions for object evolution.

In the O/O databases literature, the problem of object evolution can be classified into roughly two categories: *schema evolution*, for example modifying a class definition, and *instance evolution* – e.g., migration of an instance of one class to another.

One of the pioneering works to deal with the problem of schema evolution is the ORION system [45]. In [45] the authors present the system and establish a framework for supporting schema evolution by defining its semantics and discussing its implementation. The work mainly focuses on the subject of consistency of the methods in schema modifications, using the notion of *invariants* as constraints on the schema evolution. Invariants are for example governing the structure of the class lattice, the naming of classes, the class inheritance etc. An inconsistent schema is one that violates one or more of the invariants. In particular, the authors of [45] present a taxonomy of changes that respects the consistency of the schema under certain conditions. This taxonomy is the result of translating the invariants into groups of rules that must hold under all conditions while the schema is evolving. Reasoning on these rules by using a simple set of change operators – types of changes to a schema – allows the identification of the conditions under which changes to specific aspects of the schema (classes, attributes, relationships etc.) are consistent.

The ideas developed for the ORION system about schema consistency are generic enough to be applicable in different systems and for that reason they have been very popular within the O/O database community and beyond. The O_2 system [46] for example builds on them to propose an hierarchy of schema modifications, but makes a further

distinction of consistency into *structural* (for design time) and *behavioral* (for run time). Schema evolution is also discussed in similar terms in the GemStone system [47]. In [48] the authors provide a formal model based on the *Z* language. The proposed model is used for determining the correctness of database schema updates and is both more general and formal than the previous works. This work is using a similar approach for the evolution of services, aiming though for compatibility instead of consistency.

Another major contribution of the ORION system is the development of a versioning model for database schemas and its integration with the schema evolution model [49]. The model identifies different type of versions (transient – ad hoc private schemas, working – stable private schemas, and released – stable public schemas), incorporates a notification mechanism for communicating changes to schema consumers and allows for different levels of granularity in versioning. Approaches like [50] and [51] independently develop similar versioning models. The integration of versioning and evolution model forms the basis of this work too.

Furthermore, the development of models for schema evolution has created opportunities for cross-disciplinary research. The requirement for representing and managing the history of data objects for example led to works like [52] that use temporal databases [53] for temporal schema versioning. In a different context, [54] presents an approach on schema evolution that is aimed at supporting software engineering projects, with the emphasis on class version management and class evolution control.

For a more complete presentation on schema evolution the reader can refer to [55]. As far as instance evolution is concerned though, there is no similar comprehensive work on it. The most relevant discussion on this matter can be found in [56], [57], [58], and [59]. The migration of running instances to a new schema is a subject that has been examined extensively in the context of workflow and process management systems.

2.3.3 Workflow & Process Management Systems

The problem of workflow evolution is tightly associated with the notion of *flexibility* in workflow systems. This stems from the need of constant refinement of processes to meet the constraints, opportunities, and requirements of a fast-changing business environment. The problem again has two facets [60]: *static*, referring to the issue of modifying the workflow description, and *dynamic*, referring to the problem of managing running instances of a workflow whose description has been modified. Simple solutions like waiting for processes to finish before attempting modifications to their schemas, or aborting their execution in order to implement the changes are not usually acceptable. This makes the issue of dynamic workflow evolution very challenging.

The work on the static aspect, at least in its conception, draws heavily from the O/O databases evolution literature. An early contribution to the dynamic aspect can be found in [61], where change is formally modeled, and correctness criteria (*fault prevention*, *cancel all* and *consistency*) are introduced. Furthermore, Casati et al. [60] propose a set of primitives that allow generic modifications to a workflow while preserving the syntactical correctness for both static and dynamic evolution. Additionally, they introduce a taxonomy that

describes how running instances can be managed after a modification to the corresponding process description. The work in [62] follows a similar approach, presenting a complete and minimal set of change operations for modification of workflow instances, ensuring correctness and consistency. The focus in that case is exclusively on the dynamic aspect of the evolution, handling structural changes to running instances, but making a distinction between permanent and ad hoc modifications. The gap between the static and dynamic aspect of the problem is attempted to be bridged in [63], where version management and integrated modeling of schema instance elements are being used for this purpose.

[64] focus on verifying specific properties of workflows while they are evolving (correctness and consistency) and present solutions that ensure these properties. [65] present a survey that classifies modern workflow systems based on the operational semantics of their metamodels and the kind of correctness criteria applied to dynamic workflow changes, based on common change problems. In the same spirit, the authors of [66] discuss a series of *change patterns* and *change support features* that enable the systematic comparison of existing process management technology with respect to change support.

Given the scope of this work and the focus of most workflow- and process-related approaches on the dynamic aspect of evolution there are very few techniques that can be used in this work. The techniques of version management however present many opportunities for adoption into service evolution.

2.4 Service Evolution & Adaptation

After discussing evolution in various relevant fields the focus now is shifted to existing works on service evolution. The evolutionary strategy proposed by each work may vary though, depending on how they approach the issue of compatibility in service evolution.

On the one end of the spectrum there are approaches that do not consider whether the changes to a service version break the consumers of the service, preferring to remain as neutral as possible [67], [68], [69] and [70]. This way they leave to the developers the prerogative and responsibility of checking whether their changes break their consumers, but they also maintain a high degree of flexibility in the cases they can handle. In principle these approaches allow for multiple versions of a single service to be accessible at a time.

On the other end, there are approaches that aim to enforce non-breaking changes of services to the extent that versioning of the service description can be simply subsumed under one version, the active (i.e. deployed and running) one [71].

We distinguish between two categories of approaches for compatible evolution:

1. *Corrective* – adaptation-based approaches that actively enforce the non-breaking of existing consumers by modifying the service, and

2. *Preventive* – that attempt to confine and forbid changes that would disrupt the consumers (instead of fixing them). The compatible service evolution model developed in this work falls in this category.

The following sections summarize the most important works in service evolution and adaptation based on this classification.

2.4.1 Corrective Approaches

Corrective approaches are initiated by a change either in the context of the service (consumer requirements, laws and regulations, market dynamics, corporate strategy) or the service itself (re-design, technological advancements). They involve different mechanisms for adapting either the interface or the implementation of the service (or both) to the interoperability requirements of the service consumers.

Adaptation has been introduced in the component-based software area where adapting a component-based system means modifying one or more of its components. In practice, most components cannot be integrated directly into a system-to-be because they are incompatible. Component adaptation aims at generating, as automatically as possible, adapters to compensate for the mismatch between component interfaces and/or behavior. Numerous adaptation approaches have been proposed, see for example [72], [73], [74], [75].

In [73] the authors propose a model-based adaptation approach focusing on software interface mismatch appearing at the behavioral level. The approach takes as input the behavioral interfaces of components to be adapted, and an adaptation contract - an abstract description of the constraints which must be respected to make the involved components work together. Given these two elements an adapter protocol is generated in an automatic way. A synchronous vector method is provided for the adaptation contract language to make explicit the interactions. The work in [74] focuses on the signature level component adaptation such as names and parameters, and proposes a checking mechanism to find the signature level mismatch. In [72], the authors build an approach that uses a classification of component mismatches and identifies some patterns to be used for eliminating them. In [75], the authors address the problem of whether incompatible component interfaces can be made based on game theory by inserting a converter between them which satisfies specified requirements.

Service Adaptation

Service adaptation can be further distinguished into two categories:

1. The *interface adaptation* of services, where the goal is to solve mismatches in the signature and/or protocol of collaborating services by modifying the interfaces accordingly.

2. The *composition adaptation*, where the subject of change is the aggregation of services constituting the composite service; in this case, either the services participating in the composition are replaced by other, equivalent services, or the "gluing" connecting them is modified, or both.

[76] is an example of the first category. They present a transformation algebra that incorporates behavioral aspects and allows pairs of consumer and provider interfaces to be linked. They also provide an Finite State Machine (FSM)-based graphical notation and a mediation engine that automates and implements the expression of this algebra. In [77] the authors categorize changes that occur to the structural and behavioral representation of the service based on whether they can be automatically adapted (or they require manual intervention). In the former case they also show how to adapt the service interfaces using existing data.

The PAWS (Processes with Adaptive Web Services) framework [78] provides the methods and tools for the design-time specification of the information required for run-time adaptation of services. Each service is described as a process that is continuously optimized during its execution. The most suitable service providers in the service registry are selected based on their QoS characteristics, and they are invoked through a mediation engine that handles interface mismatches and endpoint substitution. Self-healing capabilities for detecting and repairing failures are also part of the framework.

The second category contains works like [79], [80], [81], [82] and [83], where a number of predefined adaptation scenarios are codified in the service composition (dealing with the "known unknowns").

In [79], generic service templates are transformed semi-automatically into a number of workflows per template, depending on the selection of execution path and the participating service providers, that in turn produce a number of instances per workflow. In case of failure (e.g. non-compliance to QoS constraints) the execution engine attempts to select another workflow that could be used an alternative; if this is not possible then the request is passed upstream where a new template is attempted to be made.

In [81], points of dynamic binding and rebinding defined by the designer are analyzed into cases and transformed into proxies that will deal with these cases, adapting the BPEL code accordingly to incorporate them. Each proxy is mapped to one abstract WSDL interface. Selection between candidate services for the (re)binding is achieved by means of a rule engine and on the basis of the rules defined by the designer. [82] also deals with the dynamic service selection problem using mixed integer linear programming and optimization techniques, offering the negotiation option if no feasible solution can be achieved. Their method is geared towards services supporting large processes with severe QoS constraints.

The work in [80] is based on Aspect Oriented Programming (AOP) and relies on the manual identification of mismatches (in signature and/or protocol). A set of transformation templates for the automatic generation of external specification-respecting compositions is provided that can be instantiated when a mismatch is located. AOP is also used in [83] in order to improve the flexibility of business processes expressed in BPEL. Instead of incorporating the adaptation scenarios to the process though, they attach them to the process using process-related events at the level of the execution engine. These events are intercepted during the enactment of the process and trigger (if required) alternative scenarios to be initiated. Scenarios are attached to processes using WS-Policy, so the whole approach is using only Web services technologies.

An interesting extension of the composition adaptation category is the emerging work on self-adaptation. The need for continuous reaction to change in both context and change requirements leads to the necessity for adaptation in an automatic fashion, allowing service-based applications to exhibit proactive instead of reactive capabilities [84].

Adapter Generation

Adapters are an alternative approach for preserving compatibility without modifying the service itself. The basic idea is to resolve the mismatches between the expected by the consumers and the supported by the implementation interfaces. [85], [86] for example support the (semi-)automated generation of adapters between service interfaces and implementations based on the parametric transformation of the expected and actually offered interfaces of the service.

Interface adapters can also be layered on top of each other (e.g. in mapping chains in [87], cross-stubs and custom handlers in [88] or chain of adapters in [89]) to "mask" the mismatches and maintain compatibility between providers and consumers. By using this technique, service developers deal with an ideally unique implementation endpoint that exposes multiple versions of interfaces that are mapped to each other with adapters, instead of multiple versions of the service. The maintenance cost then is moved to the consistency and efficiency of the layering of the adapters and out of the service life cycle itself.

The inverse approach for masking is advocated by [90]: a *service proxy*, that is, a single unchanging interface is exposed to the consumers and adapters are required for mapping this interface to the various implementation versions that are developed and deployed. Multiple proxies are allowed per service in the case of incompatible versions.

The concept of self-adaptation manifests also in adapter-related approaches. [89] and [91] for example discuss self-configuration techniques for enabling the automatic adaptation of the adapters themselves to changes in the context or the consumer requirements. In a similar spirit, [77] combines autonomic computing and agent technology with SOA in order to leverage the adaptability required in modern business environment.

2.4.2 Preventive Approaches

There are a number of issues with the corrective approaches with respect to service evolution. Firstly, adaptation does not necessarily happen in response to change; it may actually be the cause of change. For example, adaptation may be used for enabling the reuse of services (e.g. [76]). In that respect, adaptation is one of the means by which evolution manifests, the other being the replacement of the services used for the composition and the redeployment of a service in case of service compositions [16].

Furthermore, the application of service adaptation techniques – both for interface and composition – is not always possible without explicit manual intervention. In this sense these approaches are limited in their automation. They may be successful in preserving interoperability with a desired set of consumers, but by definition they require a number

of modifications towards this purpose. These modifications may in turn interfere with the operation of other services by the same organization in terms of resources (computational and financial) and code. Service adapters avoid this risk by not requiring the redevelopment of the service. They move however the service adaptation cost to the effort required for developing, and more importantly, maintaining the adapters. In both cases, the benefit of adaptation in a resource-centric environment like enterprise services should always be weighted first against its cost.

Finally, the majority of the corrective approaches discussed above focus on the generation of the adaptation with the goal to automate the process. In their effort to do so however they neglect to check whether the adaptation is necessary. In other words they always assume that the change that occurred to the service is not preserving the compatibility and the situation has to be ameliorated. However, this is not always true as for example discussed in [87], [88], [89]. These approaches incorporate compatibility checks before attempting to generate suitable adapters. This is a step that is missing in most of the other approaches in adaptation. For these reasons, in our approach we focus on the preventive aspect of compatible evolution.

A series of articles from the industry discuss service evolution in a preventive manner, most commonly in conjunction with service versioning [92], [93], [94], [95], [96], [97], [71], [98], [99], [100]. They all propose a common backward compatibility-oriented strategy for versioning: maintain multiple active service versions for major releases but cut maintenance costs by grouping all minor releases under the latest one. Patterns of changes that respect backward compatibility are given as guidelines, usually in the form of modifications of WSDL files (add operation, remove operation, etc.). Implementing these guidelines in the field is the responsibility and prerogative of service developers. The use of the namespace mechanism inherent in XML is advocated whenever a non-backward compatible version is required, allowing for the breaking of compatibility with the consumers. A more detailed presentation of the techniques proposed for this purpose is included in Chapter 5, where an in-depth survey on service versioning is performed.

The approach of [101] applies the same principles to managing service evolution. They define versioning of (Web) services based on the compatibility analysis of changes. The analysis is performed using a predefined taxonomy of backward compatible changes that has to be respected in order for the service version to be compatible. Furthermore they extend the service description and registry models (in the form of WSDL and UDDI resp.) with versioning information and equip the registry with a notification mechanism in order to communicate service changes to the consumers. They also show how to build a proxy for the service client that intercepts the change notifications and updates the clients without the need for redevelopment and redeployment.

A similar proposal is put forward by [102] for the VRESCo service environment. Versioning information is added to the service description and registry model, with the option for the registry to notify the consumers for changes. In addition, a more comprehensive approach in encoding the versioning history is proposed, which uses version graphs to depict the relationship and status of active and past versions of the service. Furthermore, the service proxies required for updating the service clients are using the VRESCo runtime

architecture capabilities for dynamic invocation and rebinding. Building and deploying them is for that reasons more efficient.

In [103], the authors are also using versioning information to discern between compatible and incompatible changes to services and inform the consumers accordingly. Instead however of using a service as the versioning unit, they group services into functional subsystems that contain multiple services. All services in the subsystem have the same version identifier (that of the subsystem); if one of them changes then whole subsystem will be updated into a new version. This supports a finer-grained approach where related services with possibly shared resources are evolving together. In addition, this approach promotes asynchronous updating of clients: the service consumers are always left with the decision of when – if ever – they will move to a new (incompatible) version of the service.

The approach of [104] focuses on how to assess the backward compatibility of services during their evolution. For thise purpose they rely on a set of loose guidelines to enforce compatibility and they develop a system that checks automatically whether the change is compatible or incompatible. Furthermore, by decoupling the service description elements from the service descriptions themselves they allow for service descriptions to evolve in different granularity levels. This offers further flexibility in the case of service compositions.

All of the above approaches take the very pragmatic road of providing a set of guidelines for the compatible evolution of services based on existing technologies. This dependence on guidelines however is limited in expressivity and portability in other technologies since it relies on the specifics of e.g. a particular version of WSDL. For this reason, the W3C TAG has produced a theory for the compatible evolution of languages in general [105], and XML in particular [106]. The basis of the theory is the comparison in set-theoretical terms of the languages produced and consumed by each party. Evolution is enabled by distinguishing between the *defined* language (the one that is explicitly defined in language syntax constraints) and the *accepted* language (the one that is allowed by the language constraints). Whereas the defined language must always be understandeable by the language consumer, the accepted language does not necessarily have to be understood. As long as the accepted language is a super-set of the defined language in both language producing and consuming sides the language can evolve in a compatible way. Applying this theory to different evolutionary scenarios results in a set of compatibility-preserving strategies for the evolution of language producers and consumers [11].

In [13], we propose a similar approach by abstracting from the particular technology used for the implementation of services. We present a theoretical framework that not only replicates, extends and explains the outcome of the majority of the approaches discussed above, but also provides a formal foundation on which the effect of changes to the interface of service can be reasoned on. For that purpose we present a service description model which is supported by a meta-model. The formalization of this model allows for the introduction of a versioning scheme for the artifacts of a service description. Reasoning on the compatibility of service versions is also performed on the basis of the model.

While in [13] we discuss the fundamentals of the framework for compatible evolution, we focus exclusively on the structural aspect of services. In [107], we extend the framework to the behavioral and QoS-related aspects of service description and we update it to cover

compatibility for that aspects accordingly. In addition, we introduce the notion of T-shaped changes as changes that respect the compatibility of service versions and we show how to reason on whether a change is T-shaped or not. We also present a series of change patterns that are classified as T-shaped and discuss the limitations and impact of our approach with respect to its implementation in existing technologies. The following chapters are drawing heavily from this work.

2.5 Service Change Management

A brief presentation of approaches on *service change management* follows. These approaches are focusing on how to manage change with respect to service stakeholders, domains of responsibility, propagation of change or the contractually-controlled decommissioning of versions. While outside the scope of this work, they nevertheless present some interesting solutions for the problems of version management and communicating and coordinating change that will be useful when service versioning is going to be discussed.

The work in [108] and [109] focuses on identifying the stakeholders of change and the way they affect the evolutionary process of the service. For that reason they develop a generic service reference architecture which models the different aspects of a service (implementation, execution environment, etc.). The responsibilities of the stakeholders are classified into distinct roles (provider, developer, integrator, user and broker) and for each role a domain of responsibility is assigned in terms of the reference architecture. Based on the reference architecture and the interaction of roles, they develop an impact analysis technique for notifying interested parties about changes that occurred. For this purpose they develop the infrastructure for hosting different instances of a service, together with their historical meta-data. They allow both the notification of the service stakeholders for changes based on their role (push mode) and the querying of the infrastructure by the stakeholders for change-related information (pull mode).

The SOA roles model proposed by [110] works in a similar fashion: roles that are relevant to the evolution of services and unique characteristics of SOA systems have been identified by the authors. The assignment of responsibilities to roles though has been performed empirically through the use of questionnaires instead of the prescriptive manner of [108] and the focus of the work is on verifying this assignment.

In [111] and [112] the authors present a modern take on SCM by updating it for cross-domain configuration and change management. In particular they look at SBAs that depend on services from different domains and develop a distributed architecture for service management. Each domain exposes a set of configuration items that are accessed by standard technologies (e.g. REpresentational State Transfer (REST)ful services and Atom[6] feeds). Service users are through this way able to discover and trace the evolution of the configuration items relevant to their SBAs. They also propose an approach for the establishment of a *change process* which helps transitioning a system between states while

[6]The Atom Syndication Format RFC 4287 `http://tools.ietf.org/html/rfc4287`

minimizing the impact of change. The responsibility for identifying potentially affected clients is removed from the initiator of change; instead a decentralized change coordination mechanism is used. The mechanism enforces the change and informs the consumers depending on their role (e.g. if they are simple clients or co-owners of the service). A voting protocol, similar to a two-phase commit transaction, is used for the approval and verification of a change.

The work of [113] takes a different direction for the cross-organizational management of change. They investigate different evolutionary scenarios with respect to the timing of the release of each version (e.g. sequential vs. overlapping with transitional periods) and propose different strategies for the notification of the service consumers based on each scenario. In addition they provide a technique for estimating the time needed before decommissioning a version based on information from the consumers and the dependencies with other services. For that purpose they assume the existence of contracts between service producers and consumers that clearly define the allowed "grace" period between the replacement of one version by another.

In [114] they develop further this technique by introducing a probabilistic estimator for the time required for adapting a service to external and internal changes. This estimator can be used both for improving the decommissioning times reported to the consumers and for managing the resources required for the adaptation of the service itself in a more efficient way. In a relevant effort, [115] present an approach for calculating the fitness of a service in an evolving service network [116]. The goal is to help service providers decide whether and when they should replace or decommission a particular service version depending on its performance in the network.

2.6 Service Description

Services need to be described in a consistent and universally understandable manner. In this way services can be published by service providers and discovered by service clients and developers. They can further be assembled into manageable hierarchies of composite services that are orchestrated to deliver value-added service solutions and composite applications [39]. XML, the de facto data standard for contemporary (Web) services, lacks the constructs necessary for describing the functional and non-functional characteristics of a service. For that reason, higher level standards are required for the description of the service interfaces.

Component-Based Systems (CBS) have recognized the need for such a interface description model quite early in their development. Major component frameworks like CORBA and DCOM came up with their own Interface Definition Languages (IDLs) that ensure that component providers and consumers (in SOA terminology) are able to communicate and interoperate. They allowed the definition of the operations that the component can perform, together with the input and output parameters for these operations and possible exceptions (following the example set by O/O languages like Java or C++). In a widely cited article, Beugnard et al. [117] argue that the description model of components should

in addition include a clear specification of the behavior of the component (in terms of pre- and post-conditions), its synchronizations (with respect to the sequencing and timing of method calls) and its QoS properties.

In the following we briefly review the dominating languages for the description of service interfaces.

2.6.1 Web Services Description Languages

The aptly-named Web Services Description Language (WSDL) is an XML-based specification schema for describing the interfaces of a Web service. It allows the specification of the operations, message protocols, data types for the messages payload, binding information to specific wiring protocols and address information for locating the Web service. Backed by a simple meta-model (as discussed in [10]) it specifies the syntax and grammar to describe services as a collection of communicating endpoints. It groups messages (incoming and outgoing) into operations (the processing activities to be performed by the services) and operations into interfaces. It also allows to define bindings for each interface and protocol combination, and to attach a network address to each one of these combinations.

The simplicity of the language specification, together with a very strong industrial support in terms of tools and implementations have made WSDL the dominant standard for the description of services, despite its disadvantages. WSDL for example is able to describe only the structural aspect of a service and lacks native support for behavioral and non-functional description aspects. Other specifications in the Web Service technological stack (also known as *WS-**) like Business Process Execution Language (BPEL) [118], and WS-Policy[119] have been created to ameliorate this deficit.

BPEL has recently emerged as the standard for defining and managing business process activities and business interaction protocols in terms of Web services. It is an XML-based flow language for the specification of processes and interaction protocols, supporting complex business processes and transactions. A BPEL process is a container for declaring the activities to be executed and the relationships to external partners, declarations of process data and handlers for various purposes. BPEL offers the possibility to compose Web services into a new Web service and define the business logic between each of these service interactions. Each service interaction can be regarded as a communication with a (business) partner; links to each partner are expressed as typed connectors on top of WSDL interfaces.

The WS-Policy specification defines a common framework and model for services to annotate their interface description with policies referring to domain-specific capabilities, requirements and general characteristics. Policies are expressed as assertions that manifest either as requirements and capabilities of the exchanged messages (e.g. authentication scheme, transportation protocol), or as service selection- and use-specific information (e.g. QoS characteristics). For the purposes of this work this latter facility is more important. WS-Policy is built on top of XML, XML Schema and WSDL and allows for algorithms that determine which policy alternative to apply depending on the context of the service.

Outside the WS-* stack, efforts like the Web Service Offerings Language (WSOL) and

the Semantic Web service description language OWL-S (formerly known as DAML-S) are providing description models of services that go beyond the structural description. WSOL is an XML-based language that extends WSDL by enabling the specification of multiple classes of service for one (Web) service [120]. A class of service is determined by its functional and non-functional constraints, simple access rights, pricing and relationships with other service offerings of the same service. OWL-S[7] is an OWL ontology that incorporates three interrelated ontologies suitable for the description of the operational and functional semantics, and the functional description of a service (the latter by using WSDL).

2.6.2 Other Initiatives

The SeSCE project[8] has produced a layered service representation scheme that is conceptually close to the work presented in this work [121]. A service is described in *facets*, with each facet covering a particular aspect of services. There are five types of facets supported: service description and service signatures (corresponding to the structural aspect of service representation), operation semantics and behavioral specification (covering the behavioral aspect), and QoS (equivalent to the non-functional layer). Each facet specification is contained in a different model, but all models are connected by a common meta-model for service representation.

Furthermore, initiatives like the OASIS SOA Reference Architecture [122] and the CBDI-SAE Meta Model for SOA [123] provide representation models that are better equipped for covering the various aspects of service representation. The OASIS SOA Reference Architecture aims at *a)* showing how SOA-based systems can effectively enable participants to interact with services with appropriate capabilities, *b)* participants to have a clearly understood level of confidence during their interactions with SOA-based systems, and *c)* for SOA-based systems to be scalable as required in each occasion. The Reference Architecture defines three conceptual views: service ecosystem, realization and ownership. Service description falls under the realization view.

No specific technologies are used for the description of a service as in the case of WSDL and BPEL. The Reference Architecture provides a multi-layered Service Description Model which connects the description of a service with its functionalities, stakeholders and participants. The Service Description Model informs the participants of what services exist, and under which conditions can these services be used. The service description defines or references the information needed to use, deploy, manage and otherwise control a service. This includes not only the structural and behavioral aspects of service description but also non-functional characteristics like service reachability, policies and contracts associated with the service.

The CBDI-SAE Meta Model for SOA follows a similar approach in defining the critical SOA concepts that are used as part of the CBDI Service Architecture and Engineering methodology [123]. Architectural concepts are grouped into a number of packages (tech-

[7] OWL-S: Semantic Markup for Web Services `http://www.w3.org/Submission/OWL-S/`

[8] `http://www.secse-project.eu/`

nology, organization, policy, service, business modeling, specification, implementation, solution modeling, and deployment and runtime). As with OASIS SOA Reference Architecture views, the Meta Model conceptualizes the key components of many different aspects of SOA. Service description is covered in the specification package, which, as above, aggregates structural, behavioral and non-functional aspects of the service description under one model.

What makes these initiatives very interesting is that despite being developed separately they share a great deal of commonalities between them. They all recognize the need for a model of service description that encompasses structural, behavioral and non-functional aspects. They use Unified Modeling Language (UML) [124] as the modeling and communication language for defining their models. They all are technology-agnostic, providing higher level of abstractions in modeling a service but they cover nevertheless most of the basic information provided by the WS-* specifications, providing implicit mappings to the meta-models of the WS-* standards. Based on these commonalities, in Chapter 4 we propose a model for service representation that combines the key concepts of these initiatives with the meta-models of the WS-* specifications in a formal setting.

2.7 Service Contracts

In legal terms, a contract is "an agreement between two or more parties, that if it contains the elements of a valid legal agreement is enforceable by law or by binding arbitration[9]". By expanding and modifying this notion, the term 'contract' has been used with different meanings in different fields.

The Eiffel language [125] for example explicitly codifies and enforces the obligations and benefits of objects and their consumers in a bilateral manner. The obligations and benefits are expressed as logical assertions in a pre- and post-condition format (as we already have discussed for the behavioral aspect of service representation). A class invariance mechanism ensures that all instances of an object will behave in the same manner, and an inheritance mechanism allows for the generalization and specialization of the contracts in sync with the objects they constrain.

The idea of imposing software contracts that specify the commitments and expectations of the software artifact in order to ensure its performance has been also applied to CBS. Beugnard et al. [117] for example are discussing four levels of contracts (basic, behavioral, synchronization and QoS – roughly corresponding to the three layers of service representation). In both cases of object- and component-orientation though contracts are essentially defined unilaterally – without the involvement of the consumer. The published contract in that sense is a set of terms that have to be agreed upon by the other party in order to use the service.

This originally led to the perception that WSDL files are a type of service contract (see for example [126] and [94]). Due to the limited, structure-oriented information contained in a WSDL document, the notion of contracts was extended to cover also the rules and

[9] `http://en.wikipedia.org/wiki/Contract`

conditions that need to be fulfilled by any requester wanting to interact with the service [127]. As we will establish in Chapter 4 however, all these aspects are already contained in our model of service representation. For that reason we prefer to use contracts in their bilateral agreement notion.

In this respect the work presented in Chapter 7 of this dissertation is influenced by the work on consumer-driven contracts [9]. Consumer-driven contracts incorporate to the service representation the obligations of the provider with respect to the service consumers. Contracts can express the consumer expectations through simple means like spreadsheets, or more sophisticated ones like WS-Policy statements. A similar, but not equivalent, notion is the Service Level Agreement (SLA) documents that ensure that a service performs within an acceptable range (and/or what the penalties are for stepping out of this range). [128], [129] and [130] for example discuss different approaches in forming a contract (in the form of an SLA) between service providers and potential service consumers.

For the purposes of this work, a service contractis a bilateral agreement between service provider and consumer that formalizes the details of the provisioning of service (contents, protocols, delivery process, quality characteristics etc.) in a way that meets the mutual understandings and expectations of both parties [131], [132]. A contract in this context is an intermediary between providers and consumers, expressed in the form of a service representation. Service contracts, as discussed in Chapter 7, facilitate the independent (that is, shallow) evolution of both parties at the expense of an increase in coupling and overhead.

2.8 Summary

Evolution as a concept has appeared in software engineering quite early on and has manifested in different forms while the field was moving from systems to objects, to components, and finally, to services engineering. Traditionally, evolution has been considered a part or an equivalent to maintenance. The move away from monolithic systems and towards large distributed systems that are continuously updated has refocused its scope. In this context evolution is the process of iterative change to a system, while being in a feedback loop with the system itself. The need for management of change in this process is being expressed in different ways depending on the field that it is applied.

Software Configuration Management for example, aims at controlling the software development processes for large complex systems. For that reason evolution is expressed through the coordination and support of development. The concept of versioning, i.e. keeping track of the history of software artifacts, is the basis on which this is achieved. Since it is inconceivable to discuss an evolutionary process without its historical records, the tools and techniques developed for SCM have influenced the work discussed in the following chapters to a great extent.

In a similar fashion, Object-Oriented and Component-Based systems are very useful in this discussion. O/O databases in particular offer very mature theories and techniques for dealing with the evolution of services as persistent long-lived objects. Ideas like invariance

(resistance to change) and consistency (conformance to the rules that govern the evolution) are essential for dealing with the compatibility of services. CBS, being the closest conceptually and historically to SOA, have important lessons to offer with respect to the classic techniques from software engineering that can (or not, in some cases) be applied to services. Research in that field is also warning us about potential pitfalls in reusability and maintainability due to the effort and cost of maintaining large distributed systems. Evolution in workflow and process management on the other hand, while interesting in its own sake, is largely out of scope of this work due to its focus to migration of instances of workflows and processes.

There are very few approaches that discuss service evolution outside the scope of compatibility. By contrast, there are many different ones for compatible service evolution; we classified them in corrective and preventive approaches. Corrective approaches are advocating the adaptation of the service itself or the semi-automatically (at best) generation of adapters in order to preserve the compatibility with service consumers. While very useful, most of them do not examine whether it is necessary to actually adapt the service and even worse, they do not take into account the cost of adaptation for the service provider. Preventive approaches are constraining evolution to compatible only changes, usually based on guidelines on what constitutes compatibility. The approach discussed in this work is in the latter category, but instead of relying on guidelines for compatibility it discusses a theory for reasoning on compatibility.

Furthermore, we briefly surveyed the State of the Art in two subjects that will be discussed more extensively in following chapters: service description and service contracts. For service description we established the domination of industry and academia by WSDL, which also created the need for languages like BPEL and WS-Policy to provide descriptions for non-structural aspects of the service description. The meta-models of these languages, in conjunction with initiatives like the OASIS SOA Reference Architecture and the CBDI-SAE Meta Model for SOA, form the basis for our service representation model (Chapter 4). On the other hand, for our model of service contracts (Chapter 7) we opted to diverge from the perception of contract as a formal description of the service and return to the root of the term (that is, of bilateral agreements between service producers and consumers).

Chapter 3

Running Scenario

> These gigantic, complex, interconnected technological systems overwhelm human values and defy human control. Change is possible in the system only if it does not conflict with primary technical values such as efficiency or large-scale integration.
>
> *George Basalla*

> I hate change! It's too disruptive! When things are different, you have to think about the change and deal with it! I like things to stay the same, so I can take everything for granted!
>
> *Calvin and Hobbes on the (de)merits of change*

3.1 Description of the Scenario

In order to explain our work in a practical setting we chose to use the Automotive Purchase Order Processing Scenario for demonstration purposes. The scenario is being developed and used as one of the validation scenarios in the S-Cube Network of Excellence[1] [133]. The scenario is based on the Supply Chain Operations Reference (SCOR) model that provides abstract guidelines for building supply chains[2]. SCOR is a cross-industry, standardized supply-chain reference model that enables companies to analyze and improve their supply-chain operations by helping them to communicate information across the enterprise and measure performance objectively. The SCOR model comprises of four levels of processes (scope, configurations, business activities and implementation, respectively).

This Automotive Purchase Order Processing Scenario is an example of how to realize SCOR level 3 activities using SOA-based processes for an enterprise in the automobile industry called Automobile Incorporation (a.k.a. AutoInc). AutoInc consists of different

[1] `http://www.s-cube-network.eu/`
[2] `https://www.supply-chain.org/`

business units, e.g. Sales, Logistics, Manufacturing, etc., and collaborates with external partners like suppliers, banks and transport carriers. A fragment of the scenario in Business Process Model and Notation (BPMN) notation[3] showing the interaction of the different business units of AutoInc is depicted in Fig. 3.1. For the complete BPMN model the reader is referred to [133].

The scenario is triggered when a customer or retailer submits a Purchase Order to the online order management system of the AutoInc Sales unit. The Purchase Order is verified for syntactical correctness and sufficient information. If the order verification is successful, the order is forwarded to the Customer Relationship Management unit for assignment of a treatment policy depending on the size of the order and the history of the customer with AutoInc, before it is passed back to Sales. Based on the treatment policy a pricing schema is selected and applied to calculate the cost of the order. Following on, planning of the inventory release is performed by the Enterprise Resource Planning unit, followed by the planning of the shipment of goods by the AutoInc Logistics unit. The calculated shipment cost and the pricing information calculated in the previous steps are aggregated into the final cost of the order and renegotiation is performed if necessary before initiating the servicing of the order. Payment is handled by the AutoInc Financial unit; after both shipment and payment have finished successfully the Purchase Order is closed in the system.

The scenario therefore contains a number of different activities to be performed, depicted in UML activity diagram notation [124] in Fig. 3.2. In the following we assume that the various activities in Fig. 3.2 are implemented as a series of services forming a *service chain*. Using a more "traditional" view of the chain, each business unit implements their own sets of services (sometimes by using services offered by a third party) and contribute them to the chain. The coordination of the chain for the enactment of the process is performed by the Sales unit; each unit though still maintains the control of their services since they can participate in other processes too.

3.2 The Purchase Order Processing Service

Since the scenario discussed above involves a number of services with a varying degree of complexity in their interactions with other services we opted to focus on one of them. This allows us to explain the concepts developed throughout the rest of this work based on a concrete and clearly bounded example rather than an abstract and open-ended one. It also gives the opportunity to discuss the interplay between the theoretical foundations of this approach and the technological limitations imposed by the dominant SOA-related standards.

For that purpose we chose the service supporting the "Receive purchase order" activity in Fig. 3.2, that is, the entry point to the process. We will refer to this service as PopService from this moment on. While being comparatively simple in comparison to some of the other services in the chain, PopService is subtly critical: in case of failure or

[3] http://www.bpmn.org/

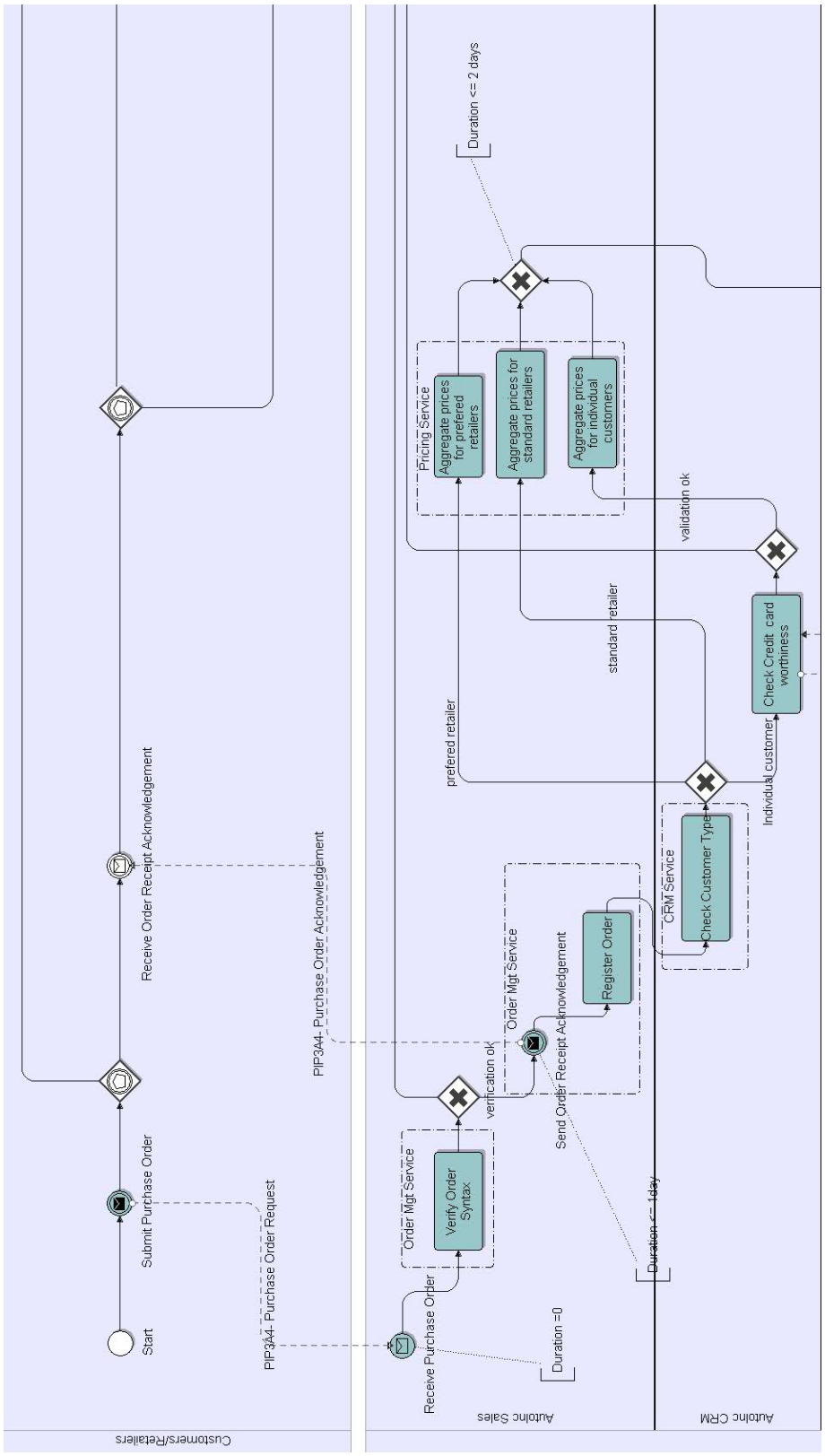


Figure 3.1: Automotive Purchase Order Processing Scenario – BPMN Model (fragment)

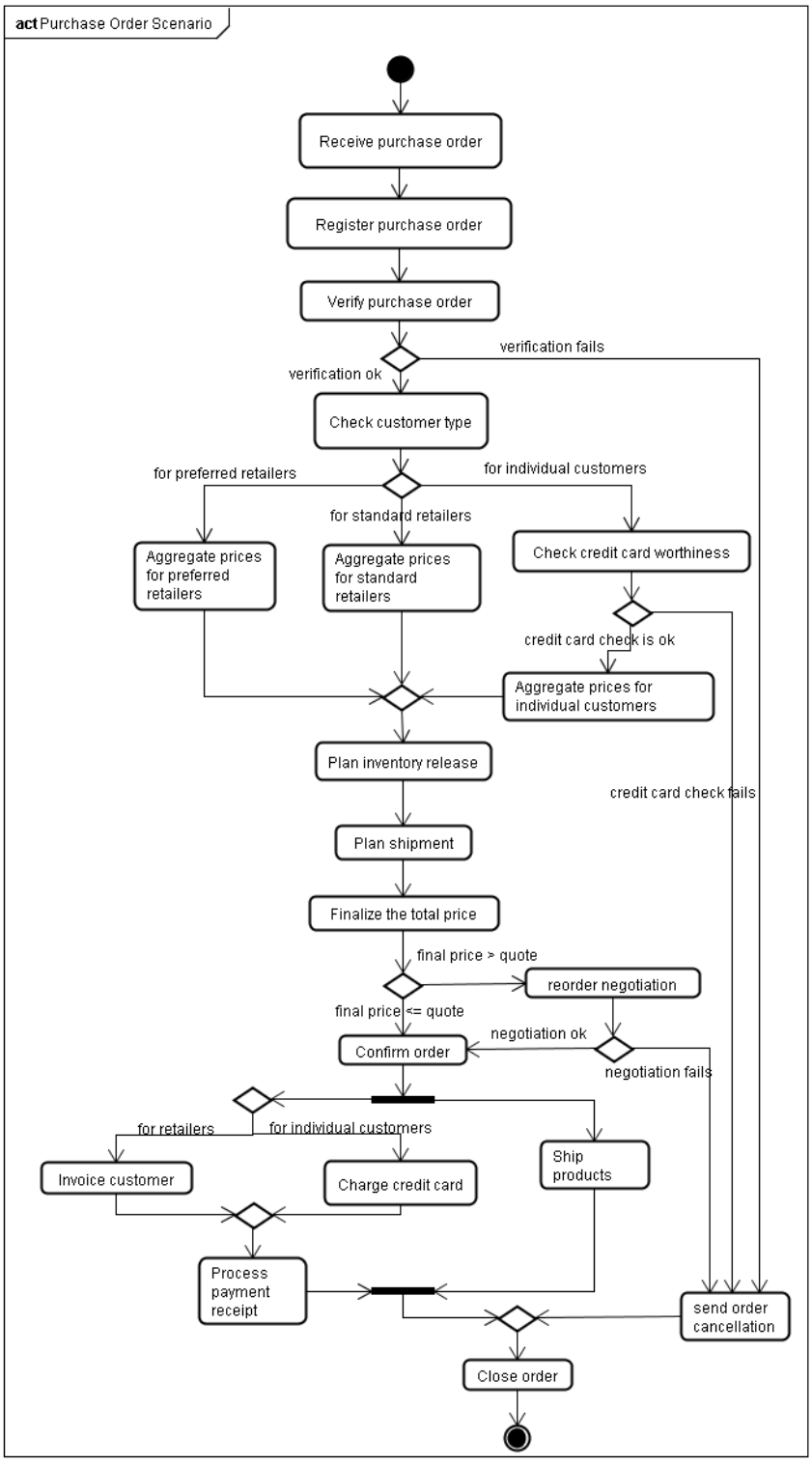


Figure 3.2: Automotive Purchase Order Processing Scenario – UML Activity Diagram

underperformance, and due to its position at the interface of the chain with the customers of AutoInc, the whole chain will either fail or underperform in turn. Other services in the chain, like for example the shipment planning, can be replaced or temporarily taken off the chain without affecting the customers. Removing the receiving of purchase orders though, even for the time it takes to adapt to a new version, has a significant impact on the customers.

Listing 3.1 contains the WSDL file for the service. Starting from its port types, PopService is communicating with its consumers in an asynchronous manner through the `receivePO` and `receivePOCallBack` operations. The actual protocol for communicating with the service is defined in BPEL in Listing 3.2 where the two-step interaction with the consumers is explicitly defined. The consumer is supposed to invoke the `receivePO` operation with the Purchase Order document, codified by the `PODocument` data type, and wait for the call back invocation `receivePOCallBack` from the service side with the acknowledgement of the order receipt.

We opted for a very simple message payload for the operations of the service which will facilitate the demonstration of the theoretical constructs we develop. Having a more complicated message payload, while not adding any particular value to the example (since the focus is not on the business modeling of the process), would only divert the attention from the application of the theory itself and towards to the superficial complexity of the scenario. The definition of the payload for the `POMessage` and `POMessageAck` messages was kept also as simple as possible for the same reasons. `PODocument`, corresponding to the Purchase Order document, is comprised of two simple strings for the order and delivery information (`OrderInfo` and `DeliveryInfo`). From these two only the order information is obligatory; in case that the delivery info is missing then the service queries the customer database of AutoInc and updates the Purchase Order with the last used address of the customer.

Finally, for the non-functional aspect of the PopService and given the absence of a widely acceptable standard for the characterization of non-functional properties we use the the S-Cube Quality Reference Model (QRM)[4] [134]. In particular, the QRM characteristics used for the definition of the PopService non-functional properties are:

- *Availability:* Availability of the service provided to customers. This is the degree of availability of the service relative to a maximum availability of 24 hours, seven days a week.

- *Latency:* Time passed from the arrival of the service request until the end of its execution/service.

- *Reliability:* The ability of a service to perform its required functions under stated conditions for a specified period of time. It is the overall measure of a service to maintain its service quality.

[4]See Chapter 4 for more information on the description of non-functional service characteristics.

Property	Value
Availability	Average 92%, minimum 80% and maximum 95% of the time
Latency	Minimum 15 secs, maximum 30 secs
Reliability	Minimum 90% across the board
Authentication	HMAC-SHA1 signature
Data Encryption	Base64Binary

Table 3.1: PopService Non-functional Properties (version 1.0)

- *Authentication:* Authentication is the process of verifying that a potential partner in a conversation is capable of representing a person or organization.
- *Data Encryption:* Refers to the algorithms adopted for protecting data from malicious access. As one algorithm may be better than another one, it also reflects the efficiency of the data encryption algorithms and mechanisms.

Table 3.1 contains the published non-functional characteristics of the PopService using the terms defined by the QRM.

In particular, latency is expected to vary between 15 and 30 seconds. Availability varies between 80 and 95% from the maximum possible availability of 24 hours, 7 days a week, with an average of 92% across the board. Reliability is at minimum 90% for the same conditions. The data encryption is using a Base64Binary XML Schema encoding [135] which uses an HMAC-SHA1 type signature for authentication.

3.3 Evolutionary Scenarios

Since the goal of this work is to study the evolution of services it is only natural to perceive PopService as subject to change. In order to illustrate possible evolutionary paths that the service can take during its life time we describe three *change scenarios*. For each scenario we provide first a high-level view of the motivation of the particular change; then we describe its (hypothetical) impact to the service and finally we show how the scenarios are affecting the WSDL and BPEL files and the non-functional properties of the service.

3.3.1 Change Scenario I

Motivation: During the operation of the PopService it was found out that too many errors originated in the handling of the delivery information. Many new customers were omitting this information and they were required to provide this information separately at a later stage. Returning customers wanted a delivery to a different address than the last used one and communicated this information when the process had already proceeded at later stages causing disruption. In addition, it was found out that due to the topology of

Property	Value
Latency	Minimum 7.5 secs, maximum 15 secs
Reliability	Minimum 81% across the board

Table 3.2: PopService Non-functional Properties – Change Scenario I

the AutoInc systems, a big part of the response time was spent in querying the customer database to retrieve the delivery information.

Impact: A new version of the PopService was designed that asks customers to obligatorily provide the delivery information along with the purchase order. Furthermore, in order to streamline and accelerate the servicing time of each order it was decided that the service will also forward the delivery information to the Logistics unit to verify that the address does not contain an error and that it points to an existing place. The result of the removal of the customer database query and the invocation of a service in the Logistics subsystem resulted in the PopService having worse reliability by 10% (due to communication faults) but better latency by 50% on average.

Outcome: The `DeliveryInfo` element has to be obligatory in the structural description of the service (Listing 3.3). The rest of the WSDL file remains as is.

```
<xsd:complexType name="PODocument">
 <xsd:sequence>
  <xsd:element name="OrderInfo" type="xsd:string"/>
  <xsd:element name="DeliveryInfo" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Listing 3.3: PopService WSDL – Change Scenario I (`PODocument` only)

The latency and reliability non-functional properties change to the values shown in Table 3.2.

3.3.2 Change Scenario II

Motivation: A new customer of PopService requests to use a synchronous communication pattern with the service for application safety reasons. Due to the amount of expected orders coming from this customer it is decided that the service should provide both synchronous and asynchronous interfaces.

Impact: The new signatures for the synchronous interaction had to be added to the existing signatures. The synchronous operation will use the same message types as the asynchronous one since they are meant to carry the same payload. Furthermore, the communication protocol of the service has to be enhanced by a new input option, allowing

the customer to decide which type of communication to use, and a new exit point to match the new communication style. The rest of the process is left unaffected.

Outcome: A new operation and wrapping port type for the synchronous communication as shown in Listing 3.4 was added to the WSDL of the service. `receivePOSync` reuses the same messages as its asynchronous counterpart `receivePO` in Listing 3.1.

```
<portType name="POPServicePortType2">
 <operation name="receivePOSync">
  <input name="poMessage" message="tns:POMessage"/>
  <output name="poMessageAck" message="tns:POMessageAck"/>
 </operation>
</portType>
```

Listing 3.4: PopService WSDL – Change Scenario II

The BPEL file of PopService had to be enhanced by a new partner link for the synchronous operation. Furthermore, the simple sequence/receive in Listing 3.2 had to be replaced by a pick activity that acts as a multiple option receive. Depending on whether the synchronous or asynchronous version of the operation was invoked, the appropriate response scheme is used (i.e. with a reply activity instead of the call back invocation for the synchronous part). These changes are depicted in Listing 3.5.

3.3.3 Change Scenario III

Motivation: Due to new regulations being implemented, every incoming and outgoing message from the service must contain a time stamp which is recorded in a separate database for auditing purposes. The regulation comes into effect after six months; until then all services in AutoInc's portfolio must comply with it. In effect, this means that previous versions of the services will be active but in "to be deprecated" status for the next six months, after which they will be replaced by their new versions.

Impact: All the message schemas of the AutoInc services (including PopService) must be updated so that they include time stamps. New versions of the services will replace the existing ones (irrespective of whether they break their consumers or not) by the deprecation date. The recording of the time stamps increases the latency of the service but not significantly, so for this reason changes to non-functional properties are not included in this scenario.

Outcome: The WSDL description of the PopService is enhanced with time stamp information for the `PODocument` and `POMessageAck` elements (Listing 3.6). A new `Information Type` element, `TimeStamp`, is created for that purpose.

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions targetNamespace="http://fnord.autoinc.com/
    PurchaseOrderProcessing"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://fnord.autoinc.com/PurchaseOrderProcessing"
     name="POPService">

 <types>
   <xsd:schema>
    <xsd:complexType name="PODocument">
     <xsd:sequence>
      <xsd:element name="OrderInfo" type="xsd:string"/>
      <xsd:element name="DeliveryInfo" type="xsd:string" minOccurs="0"/>
     </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
 </types>

 <!-- POMessage is the input message payload -->
 <message name="POMessage">
  <part name="request" type="tns:PODocument"/>
 </message>

 <!-- POMessageAck is the output message -->
 <message name="POMessageAck">
  <part name="response" type="xsd:string"/>
 </message>

 <!-- receivePO is used for invoking the service -->
 <portType name="POPServicePortType">
  <operation name="receivePO">
   <input name="poMessage" message="tns:POMessage"/>
  </operation>
 </portType>

 <!-- receivePOCallBack is used for the call back invocation by the service.
 The operation is expected to be implemented by the client. -->
 <portType name="POPServiceCallBackPortType">
  <operation name="receivePOCallBack">
   <output name="poCallBack" message="tns:POMessageAck"/>
  </operation>
 </portType>

</definitions>
```

Listing 3.1: PopService WSDL file (version 1.0)

```
<?xml version="1.0" encoding="UTF-8"?>

<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 xmlns:ns="http://fnord.autoinc.com/PurchaseOrderProcessing"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 name="ReceivePurchaseOrder"
 targetNamespace="http://fnord.autoinc.com/PurchaseOrderProcessing">

 <import importType="http://schemas.xmlsoap.org/wsdl/"
 location="POService.wsdl"
 namespace="http://fnord.autoinc.com/PurchaseOrderProcessing"/>

 <partnerLinks>
  <partnerLink name="Client" partnerLinkType="POPServiceLinkType"
  myRole="POPService" partnerRole="POPServiceClient"/>
  ...
 </partnerLinks>

 <variables>
  <variable name="PO" messageType="ns:POMessage"/>
  <variable name="POAck" messageType="ns:POMessageAck"/>
  ...
 </variables>

 <sequence>
    <!-- Wait for input from the client -->
    <receive name="ReceivePO" partnerLink="Client"
    operation="receivePO" portType="ns:POPServicePortType"
    variable="PO" createInstance="yes"/>

    <!-- Process the purchase order message -->
    ...

    <!-- Call back the client with the acknowledgement message -->
    <invoke name="SubmitPOAck" partnerLink="Client"
    operation="receivePOCallBack" portType="ns:POPServiceCallBackPortType"
    inputVariable="POAck"/>
 </sequence>
</process>
```

Listing 3.2: PopService BPEL file (version 1.0)

```
<partnerLinks>
 <partnerLink name="Client" partnerLinkType="POPServiceLinkType"
  myRole="POPService" partnerRole="POPServiceClient"/>

 <partnerLink name="Client2" partnerLinkType="POPServiceLinkType2"
  myRole="POPService"/>
 ...
</partnerLinks>

<variables>
 <variable name="PO" messageType="ns:POMessage"/>
 <variable name="POAck" messageType="ns:POMessageAck"/>
 ...
</variables>

<!-- Wait for input -->
<pick>
 <!-- If the asynchronous operation was invoked -->
 <onMessage partnerLink="Client" operation="receivePO"
  portType="ns:POPServicePortType" variable="PO">

  <sequence>
   ...
   <!-- Call back the asynchronous client with the acknowledgement -->
   <invoke name="SubmitPOAck" partnerLink="Client"
    operation="receivePOCallBack" portType="ns:POPServiceCallBackPortType"
    inputVariable="POAck"/>
   </sequence>

 </onMessage>

 <!-- If the synchronous operation was invoked -->
 <onMessage partnerLink="Client2" operation="receivePOSync"
  portType="ns:POPServicePortType2" variable="PO">

  <sequence>
   ...
   <!-- Reply to the client with the acknowledgement -->
   <reply name="ReplyPOAck" partnerLink="Client2"
    operation="receivePOSync" portType="ns:POPServicePortType2"
    variable="POAck"/>
  </sequence>

 </onMessage>
</pick>
</process>
```

Listing 3.5: PopService BPEL – Change Scenario II

```
<types>
  <xsd:schema>
   <xsd:complexType name="PODocument">
    <xsd:sequence>
     <xsd:element name="OrderInfo" type="xsd:string"/>
     <xsd:element name="DeliveryInfo" type="xsd:string" minOccurs="0"/>
     <xsd:element name="TimeStamp" type="tns:TimeStamp"/>
    </xsd:sequence>
   </xsd:complexType>
   <xsd:simpleType name="TimeStamp">
    <xsd:restriction base="xsd:dateTime"/>
   </xsd:simpleType>
  </xsd:schema>
</types>

<message name="POMessage">
 <part name="request" type="tns:PODocument"/>
</message>

<message name="POMessageAck">
 <part name="response" type="xsd:string"/>
 <part name="timestamp" type="tns:TimeStamp"/>
</message>

<portType name="POPServicePortType">
 <operation name="receivePO">
  <input name="poMessage" message="tns:POMessage"/>
 </operation>
</portType>

<portType name="POPServiceCallBackPortType">
 <operation name="receivePOCallBack">
  <output name="poCallBack" message="tns:POMessageAck"/>
 </operation>
</portType>
```

Listing 3.6: PopService WSDL fragment – Change Scenario III

Chapter 4

Service Representation

> Where wast thou when I laid the foundations of the earth?
>
> Declare, if thou hast understanding.
>
> *Job 38:4, as quoted by Jim Gray and Andreas Reuter*

> A map is not the territory.
>
> A map covers not all the territory.
>
> A map is self-reflexive.
>
> *Alfred Korzybski*

In Chapter 2 we presented some of the major languages for service description. We also discussed initiatives like the SOA Reference Architecture and the CBDI-SAE Meta Model for SOA that abstract away from specific technological solutions in service description. They provide higher level description models of services that incorporate different aspects of service interfaces and offer implicit mappings to the meta-models of the WS-* standards. This allows for their instantiation into concrete service descriptions if required.

These initiatives provided us with the inspiration to abstract from the specifics of a particular description language like e.g. WSDL or WSOL and try to discern what are the basic ingredients in service description. For that purpose we went through the meta-models proposed by these initiatives and combined them with the meta-models of WSDL and BPEL in order to identify and model their common elements. The result is a *service representation model* first introduced in [13] which was consequently updated and will be presented in the rest of this chapter. In order to establish a grounding between the WS-* technologies and our model we provide mappings between the elements of our model and the constructs of WSDL, BPEL and WS-Policy (for the structural, behavioral and non-functional aspects of service representation, respectively).

In the following sections we start by discussing informally the basic concepts of the model and the connections to existing standards and technologies. We then proceed to

formalize the service description model into a formal notation suitable for the representation of services. This notation forms the basis for the versioning model and the service compatibility theory that is presented in the chapters that follow.

4.1 Abstract Service Description Model

In order to represent a service in our model we use the notion of Abstract Service Descriptions (ASDs). Each ASD is a representation of the interfaces of a service and respects a particular meta-model depicted as a UML class diagram [124] in Fig. 4.1. This meta-model divides constructs in three layers: a structural, a behavioral and a non-functional layer. The ASD Meta-model aggregates information from the service description meta-models previously discussed and acts as a foundation from which all possible service descriptions can be generated, or, alternatively, can be validated against.

The building blocks of the ASD model are called *elements* – informational constructs representing the building blocks of the service. Each element may have one or more *properties* defining its purpose and role in the ASD, and *attributes*, i.e. variables that hold instance-specific information like the currency to be used in a particular transaction. Each property has a predefined *property domain* of allowed values. Elements are connected to each other with *relationships* signifying the syntactical and semantic dependencies between them. Elements and relationships are collectively referred to as *records*. Fig. 4.1 shows the classes of elements, called *concepts*, and their relationships. It also contains the defined property domains out of which the properties of the elements are drawing values, denoted as enumerations in UML notation.

In the following we look into each of the layers in the ASD Meta-model and explain the elements, their relationships, and their purpose in the ASD model.

4.1.1 Structural Layer

The structural layer of the ASD, depicted in the lower part of Fig. 4.1, contains the method signatures and their message parameters required for the interaction of the clients with the service. In particular, it includes the following concepts:

- `Information Type` is a wrapper for the XML Schema[1] complex and simple data types that are used as parts of the message exchange. For representing simple data types, each `Information Type` contains the `valueType` and `valueRange` properties. `valueRange` expresses the allowed range of values for each `Information Type` (or `N/A` if one is not defined). `valueType` belongs to the `DataType` property domain which contains the usual simple data types from XML Schema like `int`, `double`, `string`, etc. The `document` value is used for complex data types. Since complex types may contain both simple and other complex types, the actual content of the complex types is expressed through the (optional) reflexive association relationship

[1] XML Schema version 1.1 `http://www.w3.org/XML/Schema`

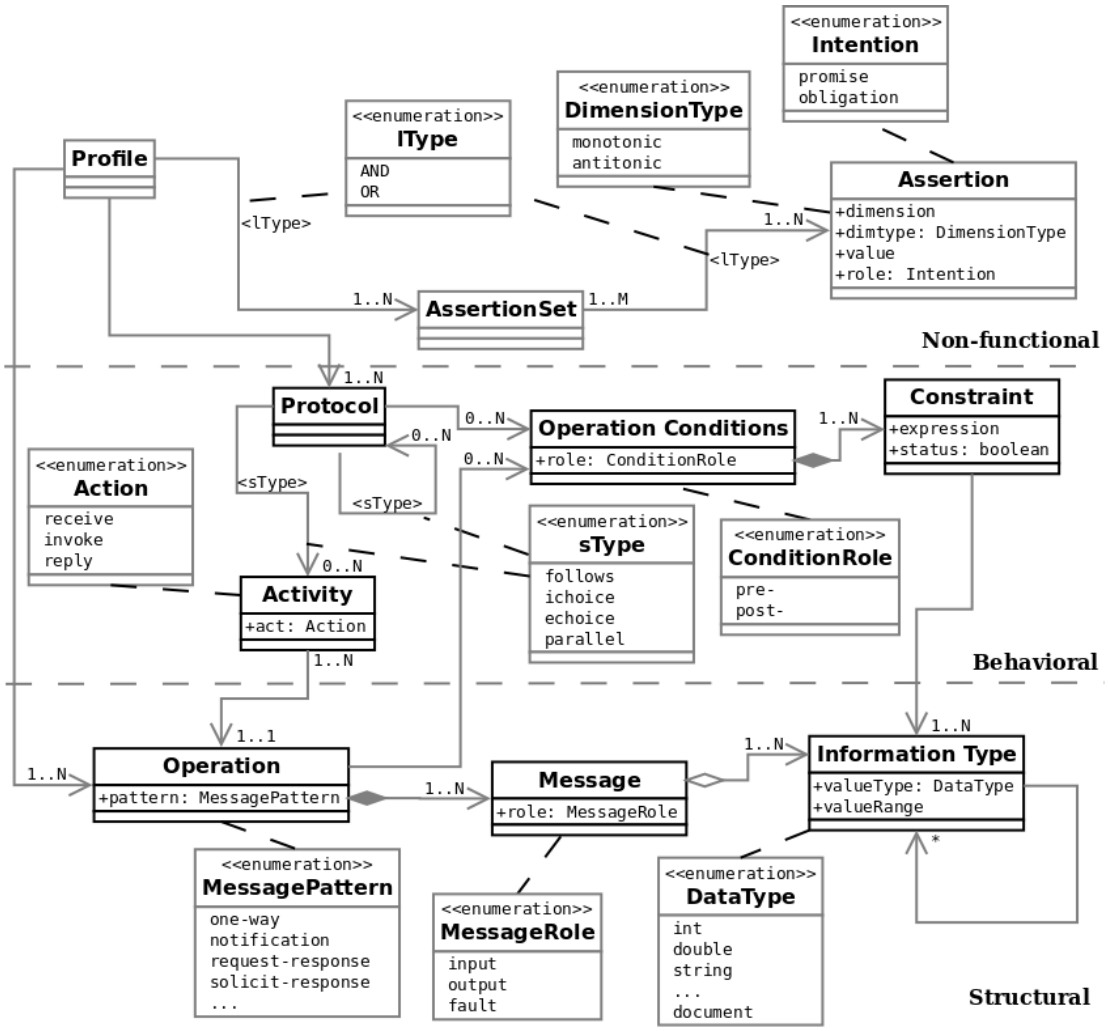


Figure 4.1: The ASD Meta-model

with other `Information Type`s. No information is explicitly stored about the particular structure of the type (e.g. the sequence of the nested elements inside a complex type).

- `Message` corresponds to the WSDL message and message part elements. It is the container of the message payload and for that purpose its property `role` draws values from the `MessageRole` property domain. `MessageRole` contains the three basic roles that a message can play in an interaction with a consumer: `input`, `output` and `fault` (that is, an exception-like output). A `Message` must contain at least one `Information Type` for "storing" the message content.

- `Operation` represents the basic interaction point of the clients with the service in the form of a discrete functionality to be performed. It contains one or more `Message`s, as defined by the semantics of its `pattern`. The `MessagePattern` property do-

main in Fig. 4.1 contains the four interaction patterns with a service (`one-way`, `notification`, `request-response` and `solicit-response`) as defined in WSDL 1.* [39]. Each interaction pattern binds the number and properties of the `Messages` it is related to. `request-response` for example would mean that the `Operation` would be connected to (at least) two `Messages`, one with property `input` and one with `output` (and optionally one with property `fault`). More powerful interaction patterns, or even customly defined ones, as provided by WSDL 2.0 and discussed in the Adjuncts section of the specification[2] can also be used here, as long as their semantics are reflected accordingly in the relationship of the `Operation` with its `Messages`. A 'robust-in-only' message pattern for example would have signify that there will be exactly one `Message` of with property `input` and so on.

While the ASD Meta-model we presented in [13] contained also an `Endpoint` concept in order to model service binding endpoints, the version of the Meta-model that will be used in the rest of this work does not contain it. This is purely for reasons of simplifying the conversation and presentation of the examples and avoid complications due to the interplay between service description and service deployment.

4.1.2 Behavioral Layer

The behavioral layer contains the records describing the perceived behavior of the service in terms of *exchanges of messages* grouped under service operations, and the *conditions under which message exchanges* may occur.

A number of different techniques have been proposed for describing and reasoning on the exchange of messages, such as business protocols based on finite state machines [136, 137] or deterministic finite automata [138], formal languages like TLA$^+$ [139], communication action schemas [76], workflows [85], automata [140, 86], timed protocols [141], [142] and Calculus of Communicating Systems (CCS)-like constructs [143], [144]. To define the conditions under which legitimate message exchanges may occur, a notation for the behavioral description of services (called *(behavioral) contracts*), which is very similar conceptually to our approach for describing a service, has been proposed in [144]. For that reason we rely on that work for the definition of behavioral description and show how the necessary constructs for applying it are incorporated into our model.

Behavioral contracts σ under [144] use three operators: *continues with* ".", *external choice* "+" and *internal choice* "$\oplus$". The behavioral contract $\sigma_1 = a_1.a_2$ means that after action a_1 is performed then it is followed by action a_2. $\sigma_2 = a_1 + a_2$ signifies that the external party (the service client) chooses which action to perform (a_1 or a_2 but not both) whereas for $\sigma_3 = a_1 \oplus a_2$, it is the service that decides which action is to be performed. Furthermore, actions are distinguished to input (to the service) denoted by a simple action a and output type (from the service to the client) actions denoted by barred actions $\overline{a}$. If not specified explicitly it is assumed that an action can be either input or output type.

[2] WSDL Version 2.0 Part 2: Adjuncts `http://www.w3.org/TR/wsdl20-adjuncts`

The middle layer of the ASD Meta-model in Fig. 4.1 contains the following corresponding concepts:

- `Activity` for actions a_i. `Activity` defines a specific type of action to be performed on the basis of an operation. It corresponds to the basic BPEL activities (`invoke`, `receive`, `reply`), as defined in the `Action` property domain. Each `Activity` is related to one `Operation` (in the structural layer), signifying that the activity is to be performed on this functionality. `Activities` with `receive` property are of input type, whereas `Activities` with property `invoke` or `reply` are of output type.

- `Protocol` for behavioral contracts σ. A `Protocol` maps to the structured activities of BPEL (sequence, pick, if and flow). The actual content of the protocol is defined through its `sType` stereotyped relation with an `Activity` or another `Protocol`, or both. `sType` has possible names `follows`, `eChoice` and `iChoice` corresponding to the operators ., $+$ and $\oplus$, for the sequence, pick and if activities respectively. The BPEL flow activity is given by the interleaving of all possible actions that can be performed as part of each (sub)protocol; as a shorthand for this interleaving the $\|$ operator is added to the basic operators by [144] and mapped to the `parallel` property in the `sType` here.

Dealing with *operational pre- and post-conditions* in order to represent the conditions under which the message exchanges can occur on the other hand is more straightforward. Towards this goal we update the classic extension of behavioral specification by [145] and [125], which describes the behavior of an object in terms of pre- and post-conditions. The conditions are expressed in the same manner as [146] as relatively simple logical expressions like $pre.elems \neq \{\}$ denoting a non-empty list of input elements. For the purposes of this discussion we will assume that these coditions are codified as groups of expressions that must be in a specific (boolean) status. The ASD Meta-model contains the following concepts required for this purpose:

- `Constraint` elements allow for the definition of specific conditions to be satisfied. Each `Constraint` is defined by a logical `expression` in simple string format, and the `status` that the expression must hold when evaluated (that is either true or false).

- `Operation Conditions` group `Constraints` together and define whether they are to be used as `pre-` or `post-`conditions as defined in the `ConditionRole` property domain for protocols (in the same layer) or operations (in the structural layer).

4.1.3 Non-functional Layer

The non-functional layer consists of QoS characteristics in the forms of assertions that are associated with evolving services. As pointed out by [121], there exists no standard for specifying the QoS attributes of (Web) services. For that reason approaches like WSOL

[120] and Q-WSDL [147] extend WSDL with QoS information that is described based on a predefined ontology of QoS dimensions. While early works like [148], [149] and [150] investigate the possibilities for representation of the non-functional service aspect, the lack of a commonly accepted standard for QoS description has forced researchers into ad-hoc QoS representation solutions based on the requirements of the application they are discussing (cf. [129] and [130]). The reader is further referred to [151] for an in-depth survey of the various efforts on service quality description.

In order to address this particular lack of standards, the S-Cube Quality Reference Model (QRM) [134] documents, consolidates and aligns the definitions of quality characteristics from diverse domains (service engineering, software engineering, business process management and grid computing) in the form of a quality taxonomy. It aggregates quality characteristics into categories like performance, dependability, security, data-, configuration- and infrastructure-related quality, usability, cost, etc. It identifies specific dimensions for each category, providing for an hierarchical organization of the QoS characteristics. Performance for example contains response time and throughput; latency is a type of response time, and execution time and queue delay time are forms of latency.

For the purposes of this work we focus on *ordinal QoS dimensions* [130] in the S-Cube QRM, i.e. QoS dimensions whose values can be ordered according to some predefined criteria. The ASD notation can also be used to express non-ordinal dimensions like security [135] or privacy [152]. The theory developed in the following chapters for reasoning on the compatibility of service versions though requires the ordinality of the records and for that reason we do not consider them in the following discussion.

More specifically, in [153], we adopted a simplified version of WS-Policy [119] for the description of QoS-related expressions. The concepts for these elements and their property-domains are depicted on the upper layer of Fig. 4.1:

- `Assertion`s contain statements about the acceptable and expected value ranges of ordinal QoS dimensions like availability, response time, throughput etc. Each `Assertion` contains for this purpose a `value`, holding the defined value range (e.g. between 95% and 99% of the time for availability). Each `value` refers to a specific `dimension` that has a `dimtype` in the `DimensionType` property domain, denoting its behavior with respect to the ordering of its values. `Monotonic` dimensions order their values with increasing order; `Antitonic` order them in decreasing order.

  Furthermore, each `Assertion` has a `role` property, drawing from the `Intention` domain[3]. An assertion can be a `promise` by the service to respect the stated value range (for each dimension), as for example in the case of promised availability. Otherwise, the assertion is an `obligation` that the service is expecting to be (externally) fulfilled, as in the case of a price to be paid per invocation of the service, or a particular authentication mechanism to be used. An `Assertion` (roughly) corresponds to a policy assertion of WS-Policy.

[3]The `Intention` domain here is used to denote the intended role of each assertion; in that sense it deviates from the Requirement Engineering definition of intention as a goal to be fulfilled as e.g. in [146].

- `AssertionSet`s, like policy alternatives of WS-Policy, organize `Assertion`s by grouping them into simple (non-nested) logical expressions through the `lType` stereotyped relationship. While WS-Policy allows for two types of logical expressions (exactly one or all), we prefer to use conjunctions and disjunctions for `lType` since they have more clear semantics. Nevertheless, the `lType` property domain that has values `AND` and `OR` in Fig. 4.1, can be extended accordingly to cover the logical expressions of WS-Policy.

- `Profile`s group `AssertionSet`s together is a similar manner using `lType` and allow for alternative sets of assertions as required for example in differentiated QoS profiles [154]. In that sense they are equivalent to policies in WS-Policy and they can be assigned to protocols (in the behavioral layer) or operations (in the structural layer).

4.1.4 Summary

Table 4.1 summarizes the previous presentation, organizing the ASD records according to the layer they belong to and mapping them to the corresponding WS-* stack specification artifact. An ASD record is either an element or a relationship. "N/A" entries signify that a respective artifact for the record is not available in the WS-* stack. The lack of such mapping does not affect the overall model since they can either be omitted or deduced from the other mappings in a trivial fashion.

Since WS-Policy expresses policy assertions and alternatives directly as logical expressions using the `wspolicy:All` and `wspolicy:ExactlyOne` constructs in Table 4.1 we can only show an indirect mapping with `Assertion` and `AssertionSet`, respectively. While there are such similar aspects of the description languages that are not covered by the mapping (e.g. organization of operations into specific port types or event handling) the provided concepts are more than sufficient for modeling the basic description constructs for service interfaces. A similar mapping as in Table 4.1 can be produced to other service description models as the SeCSE model.

4.2 Formalization of the ASD

The following section presents the theoretical aspect of the ASD notation. A formal specification notation based on type theory [155] is used for this purpose. The discussion on the foundation of the formalism is based on the structural layer following [13] before being extended to cover also the other layers. The POPService defined in the previous chapter is used for illustrative purposes.

4.2.1 Structural Layer

An ASD consists of *elements* and their *relationships*, formally defined as follows:

ASD layer	Record	WS-* Artifact
Structural	`Information Type`	`wsdl:types`
	`Message`	`wsdl:message` `wsdl:part`
	`Operation`	`wsdl:operation`

Behavioral	`Activity (invoke)`
	`Activity (receive)`
	`Activity (reply)`
		`bpel:invoke`
		`bpel:receive`
		`bpel:reply`

	`Protocol`	N/A

	`sType (follows)`
	`sType (parallel)`
	`sType (eChoice)`
	`sType (iChoice)`
		`bpel:sequence`
		`bpel:flow`
		`bpel:pick`
		`bpel:if`

	`Constraint`	N/A
	`Operation Conditions`	N/A
Non-functional	`Assertion`	(`wspolicy:`policy assertion)
	`AssertionSet`	(`wspolicy:`policy alternative)
	`Profile`	`wspolicy:Policy`
	`lType`	(`wspolicy:All`) (`wspolicy:ExactlyOne`)

Table 4.1: ASD records summary

Definition 1

An element e is a tuple

$$e := (name : string, (att_{i,i\geq 1} : attribute)^*, (pr_{j,j\geq 1} : property)^*)$$

A relationship $r(e_s, e_t)$ between elements e_s (the *source* element) and e_t (the *target* element) is a tuple

$$r(e_s, e_t) := (name_s : string, name_t : string, rel : relation, mul : multiplicity)$$

where:

- $name, name_s, name_t$ are the unique element identifiers of elements e, e_s, e_t respectively (of type string) e.g. $RequestMessage$.
- $(att_i, i = 1, \ldots, m)^*$ a set of zero or more generic types of *attribute*s (int, char, string, etc.) – for example $currency : string$, denoting the currency to be used in the scope of a specific message.

- $(pr_j, j = 1, \ldots, n)^*$ a set of zero or more *properties*, that is, attributes with predefined value ranges and characteristics. Properties contain information about the elements as defined by their concept and belong to a *property domain*. The `messagePattern` to be used for an operation is an example of a property domain, containing properties like `one-way`, `request-response`, etc.

- *rel* is the type of *relation* between the elements (aggregation, composition or association with the semantics defined below).

- *mul* is the *multiplicity* of the relationship, defined as $mul := [min_{crd}, max_{crd}]$ where $min_{crd}, max_{crd} \in \mathbb{N}$ (the set of natural numbers) is the minimum and maximum respectively multiplicities allowed for each member of the relationship, as denoted in Fig. 4.1.

In order to show relationships between elements we define the formal semantics of the relationships *composition*, *aggregation*, and *association* between elements x and y in the UML class diagram notation [13]:

1. *Composition c*: $\forall y, \exists! x : r(x, y) = r(x, y, c, \ldots)$; y can belong in exactly one composition relationship with x. Additionally, deleting x deletes also y (*cascading delete*).

2. *Aggregation a*: $\forall y, \exists x : r(x, y) = r(x, y, a, \ldots)$; y may participate in more than one aggregation relationships with x. Deletion of x deletes also y, but only if there are no other relationships of this type in which y participates.

3. *Association s*: $\exists y, \exists x : r(x, y) = r(x, y, s, \ldots)$; No further restrictions on the participation and the existence of y.

Extending the formalization to the other layers, as the following sections discuss, requires to add some layer-specific relationships in order to encode the semantics of the layer. These three relationship types though are sufficient for describing the dependencies between the elements of the structural layer.

In order to illustrate how the formalization works let's assume the POPSERVICE introduced in Listing 3.1 and focus on the messages and data types definition part reproduced for convenience in Listing 4.1. As indicated by Table 4.1, the purchase order document type `PODocument` and its wrapping message `POMessage` map to the ASD Meta-model concepts `Information Type` and `Message`, respectively. The following elements are therefore contained in the ASD of the service:

$$e_{pod} = (name = PODocument, valueType = document, valueRange = N/A) \qquad (4.1)$$

i.e., there are no attributes, `valueType` is 'document' and `valueRange` is undefined; in a similar fashion,

$$e_{msg} = (name = POMessage, role = input) \qquad (4.2)$$

```
<wsdl:types>
 <xsd:schema>
  <xsd:complexType name="PODocument">
   <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:schema>
</wsdl:types>

<message name="POMessage">
 <part name="request" type="tns:PODocument"/>
</message>
```

Listing 4.1: POPService version 1.0 structural fragment

We can equivalently write these elements in shorthand notation as: $e_{pod} = (PODocument, document)$ and $e_{msg} = (POMessage, input)$, respectively.

Since the e_{pod} element must contain exactly one order description item *Order Info* but it may only contain one delivery description item *Delivery Info*, the respective elements are

$$e_{oi} = (OrderInfo, string) \tag{4.3}$$

and

$$e_{di} = (DeliveryInfo, string) \tag{4.4}$$

and the multiplicities of the relationship between e_{pod} and e_{oi}, e_{di} elements must be $[1, 1]$ and $[0, 1]$ respectively. The relationships of the e_{pod} element are therefore written in this notation as:

$$r(e_{pod}, e_{oi}) = (name_s = PODocument, name_t = OrderInfo, rel = s, mul = [1, 1]) \tag{4.5}$$

and, in shorthand:

$$r(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, s, [0, 1]) \tag{4.6}$$

Similarly, the relationship between e_{msg} and e_{pod} is

$$r(e_{msg}, e_{pod}) = (POMessage, PODocument, a, [1, 1]) \tag{4.7}$$

Expressions (4.1)-(4.7) are the ASD notation equivalent of the service fragment in Listing 4.1. The rest of the service description in Listing 3.1 can be expressed in a similar fashion using the notation explained above: e_{poack} is the `Message` element representing the response message, e_{res} is the `Information Type` element holding the message payload (a string) and $r(e_{poack}, e_{res})$ the aggregation relationship connecting them.

The notation presented above for the structural layer acts as the foundation on which the other layers are formalized. Defining the formal notation for the behavioral and non-functional layers is performed by adding layer-specific semantics to the elements and relationships of the structural layer (where necessary), as presented in the following sections.

4.2.2 Behavioral Layer

Elements of the behavioral layer are following Definition 1 without particular deviations. The basic difference in the formalization of the behavioral layer with respect to the structural layer is the special treatment of the `sType` stereotyped relationship, which corresponds to the operators $., +, \oplus, \|$ for the behavioral contracts. While the formalization of relationships in Definition 1 remains the same, the types of relationships to be used (a, c, s) has to be extended to cover also the $follows$, $eChoice$, $iChoice$ and $parallel$ relationships (resp.) between `Protocol`- and `Activity`-type elements. Activities are also marked with information whether they are input or output type.

A sequence of a message reception followed by a reply activity (that is, a synchronous communication pattern between service producer and consumer) for example in the behavioral contract notation is expressed by the contract $\sigma_1 = a_{rcv}.\overline{a_{rpl}}$. The same expression in ASD notation is:

$$\begin{aligned} e_{prt_1} &= (name = seq_1) \\ e_{rcv_1} &= (name = rcv_1, act = receive) \\ e_{rpl_1} &= (name = rpl_1, act = reply) \\ r(e_{prt_1}, e_{rcv_1}) &= (seq_1, rcv_1, follows, [1,1]) \\ r(e_{prt_1}, e_{rpl_1}) &= (seq_1, rpl_1, follows, [1,1]) \end{aligned}$$

where e_{prt_1} is a `Protocol` element and e_{rcv_1}, e_{rpl_1} are `Activity` elements. The $follows$ relationship here between e_{rcv_1} and e_{rpl_1} signifies that the activities they represent are connected by the . operator under protocol e_{prt_1}. $\sigma_2 = a_{act_1} + a_{act_2}$ is in turn $r(e_{prt_2}, e_{act_1}) = (prt_2, act_1, eChoice, [1,1])$ and $r(e_{prt_2}, e_{act_2}) = (prt_2, act_2, eChoice, [1,1])$ and so on.

Denoting a behavioral contract $\sigma_3 = (a_1.a_2) + (a_1 + a_2)$ in this notation requires three protocols: an e_{prt_1} protocol as before to denote the $(a_1.a_2)$ (sub)contract, an e_{prt_2} similarly for the $(a_1 + a_2)$ subcontract and an additional protocol e_{prt_3} with relationships $r(e_{prt_3}, e_{prt_1}) = (prt_3, prt_1, eChoice, [1,1])$ and $r(e_{prt_3}, e_{prt_2}) = (prt_3, prt_2, eChoice, [1,1])$ for connecting the two subcontracts together.

> In a similar fashion we can always map *any* `Protocol` element $\mathtt{e_{prt}}$ and its relationships to other `Protocol` elements ($r(\mathtt{e_{prt}}, \mathtt{e_{prt_i}})$) and/or `Activity` elements ($r(\mathtt{e_{prt}}, \mathtt{e_{act_j}})$) to the respective behavioral contract $\sigma(\mathtt{e_{prt}})$.

Using the mapping between BPEL and the behavioral contracts notation discussed above and summarized in Table 4.1 allows for transforming a BPEL behavioral description into an equivalent expression in the ASD notation.

For example, the protocol described in Listing 3.2 and partially reproduced in Listing 4.2 for convenience maps to the behavioral contract expression $\sigma(e_{sequence}) = a_{ReceivePO}.\overline{a_{SubmitPOAck}}$, which, following the previous presentation, maps to the ASD elements and relationships

```
<sequence>
   <receive name="ReceivePO" partnerLink="Client"
   operation="receivePO" portType="ns:POPServicePortType"
   variable="PO" createInstance="yes"/>
   ...
   <invoke name="SubmitPOAck" partnerLink="Client"
   operation="receivePOCallBack" portType="ns:POPServiceCallBackPortType"
   inputVariable="POAck"/>
</sequence>
```

Listing 4.2: POPService version 1.0 behavioral fragment

$$
\begin{aligned}
e_{sequence} &= (sequence) \\
e_{ReceivePO} &= (ReceivePO, receive) \\
e_{SubmitPOAck} &= (SubmitPOAck, invoke) \\
r(e_{sequence}, e_{ReceivePO}) &= (sequence, ReceivePO, follows, [1,1]) \\
r(e_{sequence}, e_{SubmitPOAck}) &= (sequence, SubmitPOAck, follows, [1,1])
\end{aligned} \tag{4.8}
$$

Expression set (4.8) is the equivalent of Listing 4.2 in ASD notation. `Operation` `Conditions` and `Constraint` elements and their relationships are covered by the discussion on the structural layer since they use only the basic relationships.

4.2.3 Non-functional Layer

Describing elements in the non-functional layer is performed in a similar manner as above. Again, the only modification of the foundation of the formalism is the inclusion of the `lType` relationships *AND* and *OR* in Definition 1. A relationship of this type is to be interpreted as a logical expression connecting the source element with (all) target elements.

Using the ASD notation, any element $\mathtt{e_{assert}}$ of `Assertion` type is defined as a tuple

$$\mathtt{e_{assert}} := (name : string, dimension, dimtype : dimensiontype, value, role : intention)$$

For the values of the QoS properties of the POPService as defined in Table 3.1 for example we have:

$$
\begin{aligned}
e_{assert_1} &= (assert_1, availability, monotonic, [80, 95], promise) \\
e_{assert_2} &= (assert_2, latency, antitonic, [15, 30], promise) \\
e_{assert_3} &= (assert_3, reliability, monotonic, [90, 100], promise)
\end{aligned} \tag{4.9}
$$

Since we are using ordinal dimensions we can not express the authentication and data encryption properties of Table 3.1 and for that reason they are excluded from this presentation. All assertions refer to the promise of the service to provide QoS characteristics within

the defined value ranges and for that reason all have the *promise* role. Expressing the grouping of assertions (4.9) under an assertion set (e.g. $aset_1 = assert_1 \wedge assert_2 \wedge assert_3$ in the example) is denoted in a similar manner that was used for the behavioral layer by relating the assertion elements $e_{assert_1}, e_{assert_2}, e_{assert_3}$ with the assertion set element e_{aset_1} using a conjunction:

$$\begin{aligned} e_{aset_1} &= (aset_1) \\ r(e_{aset_1}, e_{assert_1}) &= (aset_1, assert_1, AND, [1,1]) \\ r(e_{aset_1}, e_{assert_2}) &= (aset_1, assert_2, AND, [1,1]) \\ r(e_{aset_1}, e_{assert_3}) &= (aset_1, assert_3, AND, [1,1]) \end{aligned} \tag{4.10}$$

POPService offers only one profile to its consumers e_{pfl_1}, containing exactly one assertion set (e_{aset_1}):

$$\begin{aligned} e_{pfl_1} &= (pfl_1) \\ r(e_{pfl_1}, e_{aset_1}) &= (aset_1, assert_1, OR, [1,1]) \textit{ (by convention)} \end{aligned} \tag{4.11}$$

Combinations of disjunctions and conjunctions and more complex logical expressions can be denoted in the ASD notation in a similar manner. It has to be noted that by using `lType` relationships it would be possible to decompose `Constraints` from the behavioral layer into more fine-grained expressions similar to the `Assertion` elements of this layer. For this presentation though we prefer to keep them separate and simplify the discussion.

4.2.4 Formal Definition of ASD

The foundation and the extension of the ASD formalism to cover all the description layers of a service allows the formal definition of a service description artifact as:

Definition 2

The Abstract Service Description of a service s is the set

$$\mathcal{D} := \{e_i, r_j | i \geq 1, j \geq 1\}$$

of its elements and relationships for all layers. The members of $\mathcal{D}$ – either elements or relationships – are jointly called the *records* of the ASD.

By combining Expressions (4.1)-(4.7) for the structural layer, (4.8) for the behavioral, and (4.9)-(4.11) for the non-functional layer, the ASD $\mathcal{D}$ of the PopService for example comprises of the elements

- e_{pod}, e_{msg}, e_{oi}, e_{di}, e_{res}, e_{poack} for the structural,
- $e_{sequence}$, $e_{ReceivePO}$, $e_{SubmitPOAck}$ for the behavioral,

- and $e_{assert_1},\ e_{assert_2},\ e_{assert_3},\ e_{aset_1},\ e_{pfl_1}$ for the non-functional layer,

and their relationships

- $r(e_{pod}, e_{oi}),\ r(e_{pod}, e_{di}),\ r(e_{msg}, e_{pod}),\ r(e_{res}, e_{poack}),$
- $r(e_{sequence}, e_{ReceivePO}),\ r(e_{sequence},\ e_{SubmitPOAck}),$
- and $r(e_{aset_1}, e_{assert_1}),\ r(e_{aset_1}, e_{assert_2}),\ r(e_{aset_1}, e_{assert_3}),\ r(e_{pfl_1}, e_{aset_1}),$

respectively.

4.2.5 ASD Consistency

Since services are subject to change it is easy to envision situations where changes to a service leave its ASD in a logically inconsistent state. Deleting all the messages that an operation is using for example should not be allowed since then the operation would not be able to interact with the service consumers. For that reason in [13], and influenced by [45], we defined a series of *invariants* that must hold at every state of the evolution of an ASD. In ASD model terms these invariants are:

$\mathcal{INV}_1$ $\mathbb{D} \models \mathcal{D}$ *(Validity of the ASD):* An ASD $\mathcal{D}$ must always be *valid* with respect to the ASD Meta-model $\mathbb{D}$, i.e., only elements and relationships with the defined property domains, relationship type and multiplicities in the ASD Meta-model are allowed. This also includes the preservation of the semantics of the relationships, as defined in Section 4.2.1.

$\mathcal{INV}_2$ $\forall e_i \in \mathcal{D}, \exists e_j \in \mathcal{D}\ /\ r(e_i, e_j) \in \mathcal{D} \vee r(e_j, e_i) \in \mathcal{D}$ *(Reachability of Elements):* All elements must participate in at least one relationship with another element. If there are elements without any relationships in the schema then they are automatically deleted.

$\mathcal{INV}_3$ *Property Preservation:* If there is a property pr_j of the element e_s that constraints the multiplicity of the relationship $r(e_s, e_t)$ of the element and/or the properties of the related elements e_t, then this constraint must be respected at all times.

A *consistent* (or equivalently a *well-formed*) ASD has to respect these invariants:

Definition 3
An ASD $\mathcal{D}$ is called *consistent* iff it respects invariants $\mathcal{INV}_1$-$\mathcal{INV}_3$.

For example, reducing the payload of a `Message` element by deleting one of the `Information Type` elements that it is related to is considered consistent. Deleting all of the `Information Type`s that it is related to however leaves the ASD in an inconsistent state, since it violates $\mathcal{INV}_1$: this relationship must have multiplicity *at least* 1 ($1..N$ as shown in Fig. 4.1).

While so far in the discussion in this chapter there was no notion of change in the service representation, the consistency checking provides an essential tool for managing the evolution of services. It must be noted here that *all ASDs used in the following discussions are assumed consistent according to Definition 3.*

4.3 Discussion

The previous sections presented the ASD representation model as an abstraction over existing service description models. In particular, the ASD comes with a meta-model that aggregates the concepts found in widely adopted technologies like WSDL and BPEL, with the ones from higher-level description models like the OASIS SOA Reference Architecture and the CBDI-SAE Meta Model for SOA. The ASD model is not meant to provide a new language for the description of services like WSOL or OWL-S but rather to provide a technology-agnostic formalism for the representation of services.

There are two reasons that such a formalism is necessary. First of all it allows to abstract away from the specifics of the particular languages (their "syntactic sugar"). The ASD Meta-model in Fig. 4.1 for example covers both versions 1.1 and 2.0 of WSDL and the formalism introduced does not require any modification for the transition to the latest version of WSDL. Furthermore, the foundation of the formalism on a setting based on type theory allows the application and extension of a wealth of existing work from object-oriented languages theory (and beyond) that otherwise had to be reinvented. The ASD notation presented above is the basis for the service version and service compatibility models that are discussed in the following chapters.

4.4 Summary

A technology-agnostic representation model of services allows us to abstract away from the details of a particular service description language. Furthermore it provide us with the basis on which a theory can be built for reasoning on the evolution of services. For this purpose in this chapter we developed the Abstract Service Description (ASD) representation model. Each ASD is a representation of the interfaces of a service and is composed of elements and relationships that together are collectively known as records. Classes of elements and their relationships are depicted as concepts (and their relationships) in the ASD Meta-model (Fig. 4.1).

The ASD Meta-model spans three layers of service description: structural, behavioral and non-functional. The structural layer contains the service signatures, the behavioral the service protocols and the operational constraints, and the non-functional the advertised QoS characteristics of the service. In the previous sections we defined the concepts and relationships in each layer by using UML notation, documenting their semantics and providing mappings to constructs of WS-* languages. We then defined a formal notation for the ASD model that is representing elements and relationships as tuples of labeled

records and defined ASD consistency in this notation. This will allow us in Chapter 6 to apply and extend the type theory to ASD models in order to control the evolution of services. Towards this goal, in the following chapter we add to the ASD models versioning capabilities.

Chapter 5

Service Versioning

> Normally, nothing changes except our understanding of what may be involved with the change.
>
> *Robert S. Arnold and Shawn A. Bohner*

> Change is not defined in a sequence of succeeding frames, but in a matrix of frames that each occupy the same space and moment.
>
> *Balthasar Holz*

The previous chapter presented a formal model for the representation of service interfaces. Each Abstract Service Description (ASD), in the way that it was defined, is essentially atemporal, that is, outside the scope of the evolutionary process. Even though there may be more than one ASDs for a service that differ in one or more elements or relationships, there is no connection between them and no way to discern the process that created them. Furthermore, a service consumer is not able to perceive a service change only until after a conflict has occurred.

The field of Software Configuration Management (SCM) has, as we discussed in Chapter 2, developed the notion of *versioning* for these purposes. Versioning is the fundamental block of SCM and consists of keeping a historical record of software artifacts as they undergo a series of changes. This chapter examines at a greater depth what SCM techniques and theories have been developed for versioning and how they are relevant to *service versioning*. Following on, it examines in depth the existing approaches in service versioning as discussed briefly in Chapter 2. The purpose of this examination is to evaluate these approaches with respect to how sufficient they are for supporting the recording of service evolution. Based on the conclusions of this survey we present a formal model for the versioning of ASDs which enables the discussion in the next chapter on the compatible evolution of services.

5.1 Versioning in SCM

A standard practice in Software Configuration Management is to issue unique Version IDentifiers (VIDs) each time an artifact changes in order to be able to identify it. Capturing the relations that may exist between uniquely identified artifacts in a structured way is the purpose of the version control function of all SCM systems [30]. Following the definitions of [32], this information is organized into *version models*. A version model defines the artifacts to be versioned, the versions' identification and organization, and the operations for retrieving existing and constructing new versions. Software artifacts and their relations constitute the *product space* – while their versions are organized in the *version space*. A specific version model is characterized by

- the way the version space is structured,
- the decision of which versions of the objects are accessible externally (from the consumer's point of view) or internally (for development purposes),
- the relationships connecting the version spaces for different artifacts, and
- the way the reconstruction of old and new versions is supported.

A version v of an evolving artifact[1], also called a *versioned artifact*, represents a particular state in its evolutionary process. Every version is characterised formally by a pair $v := (ps, vs)$, where ps and vs denote a state in the product space and a point in its version space, respectively. Each version is uniquely identified by a VID that is usually generated automatically. Depending on the approach used, VIDs can be unique numbers, strings or complex expressions. The difference between two artifact versions is called their *delta*. Delta size can range from very small to very large depending on the amount of changes that were applied to the evolving artifact.

Version control functionality depends on the definition of set V of all versions $v_i, i \geq 1$ of an artifact. Historically, there are two options for defining V: either by extensional or by intensional versioning [32]. *Extensional versioning* means that V is defined by enumerating its constituents, that is $V := \{v_i \mid i \geq 1\}$. Each time a change occurs to a version v_i of an artifact, a new version v_{i+1} is created and added to its versioned space V. In *intensional versioning*, instead of enumerating the different versions, V is defined by predicate: $V := \{v \mid c(v)\}$. The predicate c defines the constraints to be satisfied by all members of V and a particular version v is constructed in response to a query q. While the versioning approaches appear to be orthogonal, in reality they can be combined into an integrated version model that assigns specific queries to versions so that either way of retrieving versioned artifacts can work [32].

Classic SCM systems group related versioned artifacts into sets called version groups or version graphs and manage the evolution of these sets [31]. The items inside a version

[1] An artifact can be a file, a class definition, a piece of documentation, a configuration script or any other software item that is used for the building and running a software system. The actual granularity of version control may vary, depending on the scope and purpose of the system that implements it.

group are organized into directed acyclic graphs, with each arc representing a successor relationship. Typically three types of successor relations exist:

1. *Revision-of*, recording sequential or historical lines of development,
2. *Variant-of*, recording parallel or simultaneously active versions of the same item,
3. *Merge*, indicating the convergence of the variants into one version.

This scheme is also known as the classic revision/variant/merge version model that appears in many version control systems such as CVS and Subversion. Calculating the deltas for two given versions, also known as the *diff* operation, can be performed in many different ways, from comparing lines of code to more sophisticated semantics-based comparisons. This information is critical for merging versions in a version group. Despite the fact that there are many advanced techniques for this purpose, commercial SCM systems have been reluctant in adopting them since they prefer to remain neutral with respect to the types of artifacts to be versioned [31].

Change set versioning is an alternative, more modern, version model. While in the classic versioning model the first-class citizens are the versioned artifacts and the changes are derived by calculating their deltas, in change set versioning it is the changes that are the first-class citizens. Sets of changes in terms of deltas to a baseline version are stored, usually in the form of a version tree, instead of recording sets of versioned items. If required, a version of an artifact is reconstructed from the change set. The two version models are therefore essentially implementing the two different approaches in representing the version space (extensional and intensional versioning, respectively).

They both come with their own advantages and disadvantages: change set versioning for example is very flexible with respect to handling multiple changes across multiple artifacts. Grouping the changes independently of where they occur allows for a more efficient handling of multiple versioned artifacts since there is no need to traverse the product space in order to identify the relations between multiple versioned artifacts (as the classic version model would require). Additionally, it is very easy to build new, not previously envisioned and designed versions out of possible combinations of change sets. Nevertheless, not all versions produced this way make sense due to possible overlaps and conflicts in the deltas that result in irregular artifacts. Furthermore, it is not always possible to combine deltas; closed and proprietary binary objects for example have to be excluded from this process. Finally, the change set approach does not scale gracefully with the number of artifacts and changes.

The classic version model on the other hand, while much less flexible and more demanding in terms of storage space for all the versioned artifacts, has been much easier to use. In combination with its scale-free behavior with respect to the number of artifacts and changes it became the standard model for industrial systems. A compromise of sorts was eventually reached when realized that the functionality of change sets can be incorporated into classic SCM systems by using their diff and merge facilities, with the artifact type to be compared as a parameter [31]. This idea has proven very popular with the contemporary revision control systems.

There are many works, both academic and industrial, focusing on the structure of the version space, the relations between versioned artifacts and the retrieval of versions from the version space. There hasn't been much investigation though on the separation between publicly and privately visible versioned artifacts. Most approaches do not even consider this as an issue, assuming it can be solved by using some sort of access control on the version control system. For service-oriented systems though, where implementation is cleanly separated from interface definition and may even be developed and owned by completely different stakeholders, this is an essential feature. For that purpose in the following we survey existing approaches in service versioning – with an emphasis on the version model for service interfaces.

5.2 Survey of Existing Approaches

The goal of this section is to highlight the best practices in recording the historical aspect of the evolution of services, as exhibited by the proposed techniques and design patterns from both an academic as well as an industrial research perspective. The results of this investigation, with respect to the versioning of the service interfaces are contained in the following sections.

The necessity for versioning support for the service life cycle development has been identified in a number of early industry articles (e.g. [156]). Following the separation of interface from implementation in services [39], there are two dimensions in service versioning:

1. *Implementation versioning*: versioning support for the code, resources, configuration files and documentation of a service.

2. *Interface versioning*: versioning support for the service description, i.e. the artifacts that describe the interaction of the service with its environment (e.g. definitions of data types in XML Schema and WSDL and Abstract BPEL documents), as discussed in the previous chapter.

The versioning of the service implementation is by definition an SCM issue and as such the techniques from this domain can be applied almost verbatim to it. Contemporary SCM systems, such as SVN, Subversion and Mercurial as presented in Chapter 2 and in this chapter, can be used for this purpose. Service implementation is assumed to be already in place for the purposes of the following discussion on service versioning – but as an enterprise-internal, non-visible to the consumers, service infrastructure.

While the versioning of implementation is an internal to the development process issue and has to be opaque to the consumers of the service, the versioning of the service description is on the other hand a very public affair and it comes with a different set of requirements. Versioning of service interfaces requires fine-grained control on the management of change for the service description artifacts. This level of control is not provided

by version control systems since they remain non-specific about the artifacts they manage. Moreover, version control systems require all parties involved to adopt a common versioning software solution. This assumption is not realistic for a technologically mixed environment like SOA.

In order to identify service interfaces versioning requirements we conducted a thorough investigation of the existing literature on the subject. Given the limited amount of relevant publications in the field we considered both academic as well as industrial research articles appearing in major technology and SOA outlets like IBM developerWorks[2], Microsoft's MSDN library[3], InfoQ magazine[4] and SOA World[5].

Table 5.1 summarizes and classifies a number of different approaches on the versioning of service interfaces; the different aspects of versioning and the content of the table will be further discussed in the following sections. Table 5.1 classifies the versioning of service interfaces according to three categories: the service versioning method, the service versioning strategy, and the change identification model. The different methods and strategies ensure that old and new services versions can co-exist in a well-behaved manner and will not break clients that are using them, while the change identification model defines how service changes are identified according to their pattern of interaction with the consumers.

5.2.1 Version Identifiers and Version Space

Service interfaces are exposed to the consumers of the service, who in turn depend on them for the interaction with the service. This dependence demands that any type of change to service interfaces has to be explicitly visible and understandable by its consumers. A simple change like the changing of a data type from float to integer for example may break the assumptions based on which the consumer communicates with the service [93].

For that purpose, and using widely-accepted SCM practices, versioning approaches distinguish between breaking and non-breaking changes. The former constitute *major* releases and the latter *minor* ones. Naming a version usually follows the `Major#.Minor#` scheme where the sequential major release version number precedes the minor one; version "1.3" for example denotes the 3rd minor version of the 1st major release. An alternative naming scheme uses a release date stamp instead of the sequence identifier [67].

Neither naming scheme from the proposed though provides information about the position of the versions in the *version graph* [32], i.e. whether version "1.3" has been developed e.g. in parallel with version "1.2" using version "1.1" as a baseline or whether it is an iteration on the latter. This information is stored on a higher level, as part of the developer's versioning control system, and is accessible to the service consumers most commonly via the documentation of the service (if at all). In the VRESCo approach [102] however the version graph is explicitly stored in the service registry, while in the WSDL and UDDI extension

[2] `http://www.ibm.com/developerworks/`
[3] `http://msdn.microsoft.com/`
[4] `http://www.infoq.com/`
[5] `http://soa.sys-con.com/`

Service Versioning Methods	XML namespace for major versions and VIDs for minor	VIDs as attributes	[67],[95],[98], [99]
		VIDs in service name	[95]
		VIDs in address	[96],[103]
	Versioning info in service registry	Custom version metadata	[99],[100],[102]
		UDDI tModel extension	[92],[68],[97],[99],[101]
	Combination of the above		[93],[94],[71],[69, 70]
Service Versioning Strategies	Multiple active versions	Without deprecation strategy	[93],[94],[95],[68], [71],[100],[103],[101], [69, 70]
		With deprecation strategy	[92],[67],[97],[98],[99],[102]
	One active + one version to be deprecated		[96]
Change Identification Model	Client		[92],[93],[94],[95],[68], [97],[98],[100],[69, 70]
	Notification		[67],[94],[101]
	Both of the above		[99],[103]
	Transparent		[71],[102]

Table 5.1: Approaches on service interface versioning

approach [69] the versioning graph can be reconstructed using the custom meta-data of the annotated service description files.

5.2.2 Versioning Methods

It has been a common observation throughout all the approaches examined that despite of the importance of service evolution, major Web services standards do not contain any native support for versioning. This has started to change with the discussion in the WSDL 2.0 primer [157] about versioning, compatibility and extensibility (as it will be discussed in the following chapter) but it is still true that versioning in Web services is being supported through the mechanisms offered by XML and XML Schema.

In particular, we can distinguish between the following service versioning methods:

1. New XML namespaces for each (major) version – marked with "XML namespace" in Table 5.1.

2. VIDs unambiguously naming a version.

3. A combination of the above.

Approaches that follow the new XML namespace technique intentionally break the consumers of the service by assigning a different namespace to either the service itself or to its data types that disrupts the binding of the service on the consumer side. New namespaces are therefore meant to be used only if a major version of a service is deployed. On the other hand, VIDs are used either as attributes (either in the root element of the document or in each element separately) or as part of the (endpoint) URL – or even in the name of the service itself. In the latter case the effect is the same as in changing the namespace in that it breaks a consumer using the service. The former cases require the consumers to be somehow able to process the versioning information and understand the implications of the naming scheme for their application. Both approaches usually rely on the `Major#.Minor#` naming scheme either directly as a VID or by incorporating it into the namespace itself. They are not mutually exclusive and as evidenced by Table 5.1, they can be used in conjunction for versioning control.

Listing 5.1 contains some examples of how these two approaches can be combined to serve Change Scenario I of PopService. In particular, the first two instances of element `PODocument` refer to the same namespace, stating that they are under major version 1; their minor version identifiers (1.0 and 1.1) are added as a `version` attribute directly to the element, denoting that the two versions can be accepted by any consumer bound to that namespace. The third version of the element though belongs to a new major version and for that purpose, it uses a different namespace with its minor version identifier numbering modified accordingly (2.0).

Table 5.1 furthermore illustrates that some of the approaches to service interface versioning use a service registry mechanism like UDDI [158] or the VRESCo registry [102]

```
<PODocument xmlns="http://fnord.autoinc.com/PurchaseOrderProcessing/1"
  version="1.0">
 <OrderInfo> ... </OrderInfo>
</PODocument>

<PODocument xmlns="http://fnord.autoinc.com/PurchaseOrderProcessing/1"
  version="1.1">
 <OrderInfo> ... </OrderInfo>
 <DeliveryInfo> ... </DeliveryInfo>
</PODocument>

<PODocument xmlns="http://fnord.autoinc.com/PurchaseOrderProcessing/2"
  version="2.0">
 <OrderInfo> ... </OrderInfo>
 <DeliveryInfo> ... </DeliveryInfo>
 <ns:DeliveryCode xmlns:ns="http://dne.com/DeliveryPlanning"> ... </
    ns:DeliveryCode>
</PODocument>
```

Listing 5.1: Versioning examples of PopService in XML

for storing and controlling the versioning information - either as an alternative or complementary to the XML-based techniques discussed above. For this purpose they propose the addition of versioning metadata in the service description model that the registry is using. In the case of UDDI this means that the `tModel` data structure, which contains the technical description of the service and provides a pointer to the service interface definition (i.e. a WSDL `portType`), has to be supplemented accordingly. This can be achieved by using either a simple VID as above, or with more information about the versioning history of the service - marked in either case as "tModel extension" in the table. The programmatic API of a UDDI registry has to be modified accordingly to accommodate storing and querying this information.

```
<logicService serviceKey="uddi:autoinc.com.PO">
...
<categoryBag>
 <keyedReferenceGroup tModelKey="uddi:uddi-org:serviceVersion">
  <keyedReference tModelKey="uddi:uddi-org:serviceVersioning:versionName"
  keyName="name" keyValue="v1.0"/>
  ...
  <keyedReference tModelKey="uddi:uddi-org:serviceVersioning:original"
  keyName="original" keyValue="uddi:autoinc.com.PO"/>
</categoryBag>
...
</logicService>
```

Listing 5.2: Versioning example of PopService using UDDI tModel

Listing 5.2 (adapted from [101]) shows how the versioning information can be added to an abstract service description (with key `uddi:autoinc.com.PO`) to denote different versions of the service. The assumption here is that each service version should have a different `portType`, and each `tModel` represents one of these versions by pointing to it. It is left to the service consumer to access this information in the UDDI registry and decide which version is suitable for his purposes.

5.2.3 Versioning Strategies

Depending on the goal of each approach with respect to the preservation of compatibility with consumers, the versioning strategy proposed may vary.

On the one end of the spectrum lie approaches that do not consider whether the changes to a service version disrupt the consumers of the service, preferring to remain as neutral as possible (e.g. [67], [68], [69] and [70]). In that way, they leave to the developers the prerogative and responsibility of checking whether their changes "break" their consumers, but they also maintain a high degree of flexibility in the cases they can handle. In principle these approaches allow for multiple versions of a single service to be accessible at a time.

On the other end lie approaches that aim to enforce non-breaking changes of services – to the extent that versioning of the service interfaces can be ideally subsumed under one version, the active (i.e. deployed and running) one [71]. A special case of this idea is proposed by [96] where there are two versions active at all times: the current one and the old version which will be deprecated within a given time period. A predefined URL toggling mechanism is used to identify which version is the current and which is the old one.

The majority of the approaches are located somewhere between these two ends; in principle they propose a common backward compatibility-oriented strategy for versioning: maintain multiple active service versions for major releases but cut maintenance costs by grouping all minor releases under the latest one. The cost of maintenance therefore varies in proportion to the number of active versions at a time. The creation of a major version, apart from breaking existing consumers, also increases the effort required for managing the service portfolio.

For that reason the approaches marked with *Deprecation* in the table take special interest in discussing different decommissioning strategies for non-active versions of the service. Despite the differences in the details, the common denominator is the decrease of number of active versions to the absolute minimum required to serve the service clients. Usually a grace period is given before decommissioning a service version and, depending on the change identification model used, either the clients are notified in advance or they have to "discover" for themselves this information.

5.2.4 Change Identification Model

In a similar fashion to versioning strategies, the model used to identify service changes may vary according to predefined patterns. These can be classified to one (or more in the case

of [99] and [103]) of the following categories according to their mode of operation:

- *Client*: Both non-breaking and breaking changes result in new versions and the identification of the existence of a new version is left to the consumer. The consumer is then required to adapt to the new version if necessary.
- *Notification*: The consumer is explicitly notified for the existence of a new version and asked to take action, usually within a given time period. The most common method of this type requires the subscription of the service consumers to some sort of informational service that will notify them when required.
- *Transparent*: Approaches that enforce non-breaking changes do not have to inform their consumers of changes since in theory the changes are transparent to them. In reality though, some of these approaches allow their clients to identify a new version using one of the methods above.

Some the approaches propose both the client and the notification model and they are marked in Table 5.1 accordingly.

5.2.5 Findings

In the previous sections we analyzed and presented service versioning as an essential requisite of service evolution. Different approaches, mainly empirical, were presented in order to illustrate the State of the Art in the field. The findings of this investigation are summarized as follows:

1. Naming and identifying the service versions can be achieved through the use of VIDs, either directly in the XML document(s) describing the service or indirectly in service registry metadata (or both).
2. The dependence of service consumers on the service description makes the versioning information critical for the consumption of the service. Breaking the capacity of a service client to use a service is easily performed through a change in the namespace and/or the service description.
3. The extensional versioning model is adopted by all the investigated approaches.
4. The structure of the version space is not directly available to the consumers and it can be only reconstructed from the documentation of the service or from the service registry (if available).
5. The majority of the approaches try to balance the maintenance cost of multiple versions with the necessity for supporting an as wide as possible range of clients. Parallel active versions are only advised for major releases and minor releases are to be folded into the latest version.

6. A grace period is necessary before decommissioning a service version, giving the time to the consumers of the service either to adapt or to migrate to the new version.

7. While not necessary to explicitly communicate (minor) versions to the service consumers, the option for notifying them can be useful in case of major releases in order to facilitate their migration.

The natural fit of VIDs and the ease of their use in XML (Schema), in conjunction with the XML namespace disambiguation mechanism, are more than sufficient for recording and communicating the different versions of the service to the consumers. This low-level mechanism prevailed over a more sophisticated organization of the version space that would allow to record and communicate the dependencies of the service versions explicitly.

Looking at the historical progression of the versioning techniques throughout almost a decade of articles on the subject it can been concluded that little innovation has occurred. The reasons for this rigidity can be traced to the exclusion of a versioning mechanism from the most popular (Web) services language specifications like WSDL and BPEL. The reluctance of the implementers of the standards to get locked in by specific version control schemes and the pervasiveness of revision control systems like CVS or Subversion for development-purposes versioning have dissuaded the specification bodies from pushing toward standard versioning solutions.

It is indicative of the maturing of the field though that the WSDL 2.0 specification contains a discussion on different approaches on versioning. This discussion is tightly connected with the issue of compatibility and it will be visited upon in the following chapter. In order to *a)* facilitate the presentation of the service compatibility theory we have developed, and *b)* to illuminate the connection between the practices of service versioning and the SCM versioning theories and techniques we present in the following a formal ASD version model.

5.3 The Versioned ASD Model

In the previous chapter we defined the ASD $\mathcal{D}$ as the set of all of its records (elements or relationships). Each record in the ASD of the service is able to evolve at its own rate. Assuming for example a mostly stable communication protocol between service providers and clients with evolving requirements, then operation elements will change much less often than their message payloads. As the description of a service can change, we need a way to keep track of the different versions of the ASD records. Furthermore we also require the means for uniquely identifying a particular version of the ASD. For this purpose we develop an ASD versioning model using the SCM principles presented in this chapter.

5.3.1 Versioned Abstract Service Descriptions

The ASD of a service evolves as the service itself evolves. Each record d of the ASD evolves at its own rate and the evolution of $\mathcal{D}$ is expressed through the changes occurring to its

constituent records. We need a way to uniquely identify and record the evolution of each record in an ASD; in other words we need *versioned records* and *versioned ASDs*. For that purpose let us assume the set $\mathcal{V}_\mathcal{D}$ containing version identifiers for all records in an ASD $\mathcal{D}$. For each record $d \in \mathcal{D}$, $v(d) \in \mathcal{V}_\mathcal{D}$ is its VID. The identifier can be a unique number, a release identifier, a release-specific namespace or any other combination as discussed in the previous sections. A versioned record is therefore the couple

$$v_d := (d, v(d)), d \in \mathcal{D}, v(d) \in \mathcal{V}_\mathcal{D}$$

A change to record v_d results to a new version $v'_d = (d', v'(d))$ with $v'(d) \neq v(d)$, that is, with a different VID from the previous version. For the purposes of this work we only consider linear versioning histories. Parallel versions of records (as in the case of the branch operation in SCM terms) are assumed to be able to be collapsed into a single history, assigning them suitable VIDs. While techniques like version graphs can be used to reflect the structure of the historical dependencies of each record, they are as we discussed in the previous section outside of the visibility of the service consumer. In that sense they have little added value for our purposes and they are not considered here.

Without loss of generality we therefore assume that $v'(d) > v(d)$, denoting a "later" version of the record (containing a greater release number, a timestamp that is in the future, a new namespace etc.). v''_d would denote an even later version of d: $v''(d) > v'(d) > v(d)$ and so on. An ASD enriched with this versioning information is called a *Versioned ASD*:

Definition 4

A *Versioned ASD* is the set of all versioned records

$$\mathcal{S} := \{(d, v(d)) \mid \forall d \in \mathcal{D},\ v(d) \in \mathcal{V}_\mathcal{D}\}$$

It is also possible to assign a VID to the versioned ASD itself by including its identifiers in $\mathcal{V}_\mathcal{D}$. $v(\mathcal{S})$ therefore denotes the version identifier characterizing $\mathcal{S}$. $v(\mathcal{S})$ however is just a convention for naming the versions of the ASD; $\mathcal{S}$ is actually uniquely identified by the versioned records $(d, v(d))$ it contains. In a similar spirit to versioned records, we write $\mathcal{S}'$ to denote a new version of $\mathcal{S}$. $\mathcal{S}'$ is essentially a shorthand for one of the following situations:

1. $\exists v_d \in \mathcal{S}, v'_d \in \mathcal{S}' : d \equiv d' \wedge v'(d) > v(d)$, that is, there exists at least one versioned record with a later version identifier in $\mathcal{S}'$ than its respective record in $\mathcal{S}$ (signifying a modification of the record).

2. $\exists v'_d \in \mathcal{S}' : \nexists v_d \in \mathcal{S}, d \equiv d' \wedge v'(d) > v(d)$, i.e. $\mathcal{S}'$ contains a versioned record v'_d for a record d' that is not included in $\mathcal{D}$ – meaning that a new record was added to the service ASD.

3. $\exists v_d \in \mathcal{S} : \nexists v'_d \in \mathcal{S}', d \equiv d' \wedge v'(d) > v(d)$, in similar fashion, $\mathcal{S}'$ does not contain any versioned record for a record d that is included in $\mathcal{D}$ – the record was therefore removed from the service ASD.

4. Any combination of the above.

Using these events as a high level description of the modifications to the ASD allows us to define the deltas, that is, the difference between two (or more) versions of an ASD using some basic operators.

5.3.2 Representing the Version Deltas

We will use three fundamental operators to describe the changes occurring to service descriptions: *add* for the addition of record, *del* for the removal of a record, and *mod* for the modification of the record (addition/removal of attributes or properties and so on). Combinations of these fundamental operators can be further used to produce more advanced operators like the renaming of a record. By applying these fundamental operators to a (versioned) service description $\mathcal{S}$ and for a record s we get the respective *change primitives*:

1. $add(s, \mathcal{S}) := \mathcal{S} \cup \{s\}$ (addition of record)
2. $del(s, \mathcal{S}) := \mathcal{S} - \{s\}$ (removal of record)
3. $mod(s, \mathcal{S}) := \mathcal{S} \cup \{s'\} - \{s\}$ (modification of existing record by replacement with new version of the record)

Depending on whether s is an element or a relationship, the change primitives are expanded accordingly: $add(e, \mathcal{S})$ and $add(r(e_s, e_t), \mathcal{S})$ for example signify the addition of an element e or a relationship $r(e_s, e_t)$ to $\mathcal{S}$, respectively. The evolution of services rarely occurs in simple increments and usually encompasses a number of changes to the service description that occur simultaneously. For that reason we define a *change set* as the fundamental degree of change to a service description:

Definition 5

A change set $\Delta\mathcal{S}$ is a set of change primitives

$$\Delta\mathcal{S} := \{operator(s_i, \mathcal{S}) \mid operator \in \{add, del, mod\}\}$$

that when applied to a service description $\mathcal{S}$ results in a new version of the service $\mathcal{S}'$, signified by $\mathcal{S}' = \mathcal{S} \circ \Delta\mathcal{S}$.

Corollary: It can be easily observed that among the change sets there are *classes of equivalence*. The change sets $\Delta\mathcal{S}_1 = \{add(s', \mathcal{S}), del(s, \mathcal{S})\}$ and $\Delta\mathcal{S}_2 = \{del(s, \mathcal{S}), add(s', \mathcal{S})\}$ for example when applied to a service description $\mathcal{S}$ have the same effect: the removal of an existing record (s) and the addition of a new record (s' - which can be used e.g. for renaming a record).

Versions of services can therefore be expressed in terms of the change sets that are required for reconstructing a version from a baseline (original) version, following the conventions of SCM.

5.4 Summary

The field of SCM has developed a number of techniques for recording the historical aspect of the evolution of software. Version models capture the relations existing between different versions of software artifacts. As part of the version model, a version space organizes the versioning history and relates the versions of the artifacts using their unique version identifiers. While there are different approaches on how the version space should be structured, the most successful in terms of acceptance and tool support is the extensional versioning that requires the storing each version of the versioned artifacts separately. Delta calculation and version merging functionalities are performed on the basis of these versions if required.

While SCM focuses on supporting the development of software through versioning, service versioning focuses on the identification and representation of different versions of service interfaces in order to support service consumers. This emphasis on the publicly versioned artifacts required solutions that are closer to the service representation than to service implementation. A series of industrial (mainly) and academic (to a lesser extent) articles have discussed this issue and while they differ in the details of their proposals, they also share a great deal of overlap in the employed techniques.

This uniformity of approaches, coming from different technological vendors and unrelated academic research groups, convinced us that the fundamentals of service (interfaces) versioning have been already established. Our contribution in this effort is to identify, classify and present them so that they are easily accessible. Furthermore, we established a connection between the formal model developed in the previous chapter for service description and the techniques produced by SCM. Essentially we introduced the temporal aspect in service descriptions and we showed how to identify and relate different versions of ASDs and/or of their records. This connection is fundamental in developing a theory for compatible service evolution since it enables the unambiguous identification of different versions of an ASD in the development continuum. In the discussion that follows we always use the versioned ASD $\mathcal{S}$ instead of the non-versioned $\mathcal{D}$; unless otherwise specified, every time the term ASD is used it is implied that it is versioned.

Chapter 6

Compatible Service Evolution

> Evolution will take its course. And that course has generally been downward.
>
> *J.B.S. Haldane*

> Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.
>
> *Jorge Luis Borges*

The previous chapters presented our proposal for a service representation and service versioning model. Having established the means to abstractly describe and uniquely identify a service in the version continuum, this chapter focuses on answering the next two main research questions of this work. More specifically, it defines what exactly constitutes *service compatibility* and what are the conditions that enable *compatible service evolution*. It also evaluates the presented solutions with respect to the relevant approaches on service evolution.

6.1 Service Compatibility

As discussed in the introductory chapter, compatibility is one of the terms that have been overloaded with many different meanings in the literature. We therefore need to give an unambiguous definition of the term, preferably by adopting and if necessary adapting one of the existing definitions to avoid further confusion. In the sections that follow we start with an informal discussion on compatibility that we then use to provide a formal definition of the term. Finally, we look into how the different preventive approaches for service evolution presented in the related work (Chapter 2) are supporting the compatibility of services.

6.1.1 Introduction to Compatibility

For the purposes of this discussion we will extend the definition for component compatibility given in [159] and separate compatibility into two distinct dimensions: horizontal compatibility (or service interoperability) and vertical compatibility (also known as substitutability or replaceability). More specifically:

> *Horizontal compatibility* or *interoperability* of two services expresses the fact that the services can participate successfully in an interaction as service provider and service consumer.

Horizontal compatibility manifests in that sense as a co-dependence relation between two interacting parties (services in the general case). The underlying assumption is that there is at least one *context* under which the two services can fulfill their roles. The term context here refers to a *configuration of the environment* in terms of the execution state of both service provider and service consumer, along with the *status* of their *resources*, and for a particular *message exchange history*. This assumption is implicit in the definition of the vertical dimension, and permeates all the definitions, formal and informal that follow:

> *Vertical compatibility* or *substitutability* (from the provider's perspective) or *replaceability* (from the consumer's perspective) of service versions expresses the requirements that allow the replacement of one version by another in a given context.

The combination of the two compatibility dimensions leads to the notion of *T-shaped changes* as depicted in Fig. 6.1. In the figure, the positioning of the two dimensions is illustrated by means of simple example. In the example of Fig. 6.1, overlapping hexagons denote compatible service versions. Service S_1 is horizontally compatible with S_2, meaning that S_1 interoperates with S_2 – either as a consumer or a provider or both. Similarly, S_2 is horizontally compatible with service S_3. There exist two more versions of service S_2 denoted by S_2' and S_2'', that while vertically compatible with each other (and horizontally compatible with S_3') they are nevertheless incompatible with S_2 as denoted by the gap between S_2 and S_3', S_2''. This signifies the existence of a major release of S_2 (namely S_2'), followed by a minor release of S_2' (namely S_2''), that broke the interoperability of S_2 with S_1 and S_3.

The two dimensions are therefore intrinsically interrelated leading to *T-shaped compatibility*: *substitutability and replaceability can be perceived as the property of preservation of interoperability for internalized changes to one or both of the interacting parties (providers or consumers)*. This enables referring simply to compatibility and denoting both aspects. If compatibility, either on the vertical or the horizontal dimension, or both, is achieved under *all* possible contexts then it is called *strict* substitutability/replaceability and interoperability, or strict compatibility, respectively.

Compatibility is traditionally further decomposed into *backward* and *forward*. A definition of forward and backward compatibility with respect to languages in general and message exchanges between producers and consumers in particular is given in [106]:

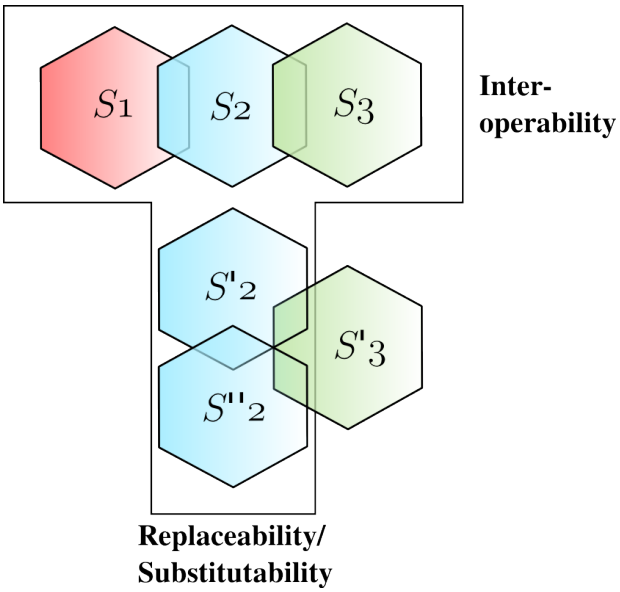


Figure 6.1: Horizontal and Vertical Compatibility

> *Forward compatibility* means that a new version of a message producer can be deployed without the need for updating the message consumer(s). *Backward compatibility* means that a new version of a message consumer can be deployed without the need for updating the message producer. *Full* compatibility is the combination of both forward and backward compatibility.

The roles of message producers and consumers can be assigned in different ways for service producers and consumers depending on the *message exchange pattern* that they use (in WSDL 2.0 terms [157]) and which defines the sequence and cardinality of abstract messages listed in an operation. For a simple request for example, the service consumers act as message producers since they generate the message that is in turn received (consumed) by the service provider; the inverse statement holds for the simple response. For patterns like request-response the roles are swapped between stages: for the request stage the roles are as in the simple request; for the response phase the service provider becomes the message producer and the service consumer the message consumer. More complicated patterns can be decomposed in simple requests and responses in a similar fashion.

6.1.2 Formal Definition of Service Compatibility

For the formalization of the compatibility between any two services we assume that each service (version) is represented in the (versioned) Abstract Service Description (ASD) notation that we presented in Chapters 4 and 5[1]. The theoretical constructs that follow can

[1]While the definitions in this section use the ASD as the representation model of the service, the formalization discussed is equally applicable to any other representation model.

be defined in the same way for unversioned ASDs $\mathcal{D}$. Since we need to be able to discern between different versions of an ASD, we will be using versioned ASDs $\mathcal{S}$ for the ensuing discussion. An ASD $\mathcal{S}$ is comprised of records s that represent the conceptual dependencies inside the service description. Records can be either *elements* or their *relationships* and they span the three layers of service description, that is structural, behavioral and non-functional.

Based on this assumption, we can take advantage of the existence of a *subtyping* relation or variations thereof, that allows us to (partially) order different records based on their characteristics for defining compatibility. Subtyping allows us to decide whether two records participate in a specialization/generalization relation and whether (under certain conditions that will be discussed in the following sections of this chapter) one record can replace another. *We will be using the notation $s \leq s'$ to denote that record s is a subtype of record s', irrespective of whether s is a structural, behavioral or non-functional record.*

In order now to formally define backward, forward and full compatibility of two service versions we will first define a distribution of the set $\mathcal{S}$ into two proper subsets $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ representing the set of records for which the service acts as a producer and a consumer (of messages) respectively [131]:

Definition 6

$\mathcal{S}_{pro}$ is the set of output-type records of a service description and $\mathcal{S}_{req}$ is the set of input-type records.

Compatibility between service versions $\mathcal{S}$ and $\mathcal{S}'$ can be defined based on this distribution as follows:

Definition 7

Service Compatibility

We define three cases of compatibility:

- Forward: $\mathcal{S} <_f \mathcal{S}' \Leftrightarrow \forall s \in \mathcal{S}_{pro}, \exists s' \in \mathcal{S}'_{pro}, s' \leq s$ (*covariance* of output).
- Backward: $\mathcal{S} <_b \mathcal{S}' \Leftrightarrow \forall s' \in \mathcal{S}'_{req}, \exists s \in \mathcal{S}_{req}, s \leq s'$ (*contravariance* of input).
- Full: $\mathcal{S} <_c \mathcal{S}' \Leftrightarrow \mathcal{S} <_f \mathcal{S}' \wedge \mathcal{S} <_b \mathcal{S}'$.

These definitions are in line with both traditional type theory [160] and with the language-producing set theory-based approach proposed in [106]. Given the fact that the service description subset $\mathcal{S}_{pro}$ represents the language produced by the service, then this definition of forward compatibility is equivalent to the informal definition given above. The same holds for the definition of backward compatibility. It has to be noted that Definition 7 is only a *sufficient* (and not *necessary*) condition for shallow changes. As we discuss in the following chapter, depending on the clients, and with some extra overhead, more leeway can be provided in the modification of the service ASD without an effect to them.

This requires however a reasoning on a per-consumer basis and an additional managerial and infrastructural overhead, that are not desirable in the general case.

Definition 7 provides the general condition for the preservation of compatibility that we accept as equivalent to confinement to shallow changes for the purposes of this chapter. Armed with this definition we can reason directly on new versions $\mathcal{S}', \mathcal{S}', \mathcal{S}'', \ldots$ of the service, comparing them on a record to record basis for deciding on their compatibility.

The implications of this reasoning will be better illustrated in the following sections where we will discuss the concepts of compatible evolution of services. In the meantime, we will first present some well-established techniques for supporting the forward and backward compatibility for (Web) services.

6.1.3 Supporting Techniques

Compatibility is a concept that is closely related in practice to versioning: almost all the approaches presented for preventive service evolution in Chapter 2 are present in Table 5.1 (containing the classification of versioning approaches) and take into account at least backward compatible changes for versioning. Backward compatibility in this context is a mechanism for distinguishing between major and minor releases: as long as the changes applied to a service lead to backward compatible versions of the service they can be considered minor releases, otherwise they are major. Some of these approaches discuss in addition techniques for forward compatibility, expressed as extensibility:

Extensibility for Forward Compatibility

Extensibility is the property of a language to allow information that is not defined in the current version of the language [106]. Extensibility therefore is a relevant notion to both versioning and compatibility: whereas versions can be either compatible or incompatible (centralized) changes to the service, extensions are by definition compatible (decentralized) additions to an existing service [161].

XML and XML Schema, the linguistic foundation for Web services, allow for extensibility through the use of *wildcards*, a mechanism for defining "open" namespaces and allowing elements from them to appear in an XML document. Listing 6.1 shows an example of an XML schema definition that uses the `xsd:any` namespace to allow for the extension of the `name` element with e.g. a middle name.

XML extensibility is in that sense an enabler of forward compatibility as evidenced by (some of) the approaches for preventive evolution. More specifically, [94], [71], [98], [99] essentially re-use the techniques summarized in [106] to show how providers can update their message schemas without breaking their consumers. The underlying assumption in all cases is that the additional data can be safely ignored during the processing of a message, without any effect on the semantics of the message. This constrains the design of the future elements significantly. The extensibility techniques discussed here can be seriously hindered though by the XML Schema 1.0 specification inclusion of the Unique Particle

```
<xsd:complexType name="name">
 <xsd:sequence>
  <xsd:element name="first" type="xsd:string"/>
  <xsd:element name="last" type="xsd:string"/>
  <xsd:any namespace="##any" processContents="lax"
   minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
 <xsd:anyAttribute/>
</xsd:complexType>
```

Listing 6.1: Example of Schema Extensibility

Change	Backwards Compatible
Add (Optional) Message Data Types to Input	Yes
Add (New) Operation (and respective Message Data Types)	Yes
Remove Operation	No
Modify Operation (Includes renaming and changing parameters, parameter order and message exchange pattern.)	No
Modify Message Data Types	No
Modify Service Implementation (As long as it has no effect on the service interfaces.)	Yes

Table 6.1: Guidelines for Backward Compatible Changes

Attribution (UPA) rule but this is corrected in XML Schema 1.1 specification (the reader is referred to [98] for a technical discussion on the UPA rule).

Preservation Guidelines for Backward Compatibility

Almost all the preventive approaches discussed in Chapter 2 for service evolution are incorporating in one form or another the notion of backward compatibility. The usual approach for defining what constitutes a backward compatible change to a service is to enumerate all possible compatible changes. This results to a list of permissible and prohibited changes, usually but not exclusively, to the WSDL document describing the service. This list reflects a combination of common sense, technological limitations and empirical findings that results into a set of best practices – guidelines to be followed and not necessarily undisputed rules. These guidelines are presented in Table 6.1, aggregating the guidelines from [93], [96] and [99]. All the changes in Table 6.1 are expressed in terms of modifications to WSDL and XML Schema elements.

In summary, the only backward compatible changes are additions of optional elements (data types or operations) or modifications of the service implementation (as long as it

does not affect the WSDL document per se). The removal or any kind of modification to an operation element is strictly prohibited, as is the modification of the message data types (with the exception of addition of optional data types).

This guideline-based approach is easily applicable and requires minimum support infrastructure and for that reason it is widely accepted, despite being very restrictive. However, it exhibits certain disadvantages, the main of which is that it depends on service developers for deciding what is compatible and what is not and acting accordingly. Even if these rules are codified and embedded into a service development/versioning tool as for example in the case of [104], they will always be limited by two factors: their dependency on the particular technology used (WSDL in this case) and their lack of a robust theoretical foundation. The first factor means that if a technological shift in implementing services occurs (in case for example WSDL 2.0 becomes widely accepted), then these rules must be recreated. The second factor on the other hand signifies that their correctness can not be formally verified (i.e. proved) but only validated through use.

For these reasons in our approach we are extending the reasoning behind the backward compatible guidelines and we enhance it by showing how these rules can be generated as the result of a theory for the control of evolution.

6.2 Type Theory for Abstract Service Descriptions

The definition of service compatibility we discussed in the previous sections relies on type theory, and in particular in the subtyping relation to distinguish between compatible and non-compatible services (Definition 7). While we briefly discussed what the subtype relation would entail with respect to the ASD formal model presented in Chapter 4, we did not fully define it. This is the purpose of this section – starting from a brief introduction to type theory and then proceeding to define subtyping for the records of each layer in a service ASD.

6.2.1 A Short Introduction to Type Theory

In the following we introduce the type theory basics using the seminal work of Cardelli and Wegner [160] as a source.

Types in the mathematical sense are sets of objects that exhibit similar behavior. Objects of a certain type are respecting the properties defined by the data type. *Type systems* impose restrictions on the objects structure and their interaction with other objects and in that sense they ensure the logical consistency of objects. Type systems usually include an *inference* mechanism that is used for deriving valid, consistent objects.

Type systems are monomorphic or polymorphic; monomorphic means that each object can be of one and only one type, whereas polymorphic means that some of the objects can be of more than one type. Polymorphism can be universal (applicable to an infinite number of types in a uniform way) or ad-hoc (i.e. it works only for a certain finite set of different and potentially unrelated objects).

The notion of *subtyping* we referred to in Section 6.1.2 is a type of parametric polymorphism called *inclusion*. It essentially boils down to the fact that every object of a subtype can be used in the context of a supertype (and is therefore *compatible* with it). A square for example is a subtype of shape (since it has all the properties of a shape, plus some extra ones – the fact that it is a polygon with four equal sides and angles). Since types are sets, subtypes can be perceived as subsets: if $\tau_1 \leq \tau_2$, that is, τ_1 is a subtype of τ_2, then it holds that $T_1 \subseteq T_2$ where T_1 and T_2 are the sets of all types of τ_1 and τ_2 respectively, ordered by inclusion (T_1 and T_2 are also called the *type lattices* of τ_1 and τ_2).

Each object is a record in the sense of [155], that is, a finite association of values to labels, denoted as a set $\{a_1 = value_1, a_2 = value_2, \dots\}$ where each expression $a_1 = value_1, a_2 = value_2, \dots$ has a type $\tau_1, \tau_2, \dots$. This property allows us to define record types as labeled sets of types with distinct labels $\{a_1 : \tau_1, a_2 : \tau_2, \dots\}$. A subtype in this context is therefore defined as the $\leq$ relation:

Definition 8

For record types τ and τ' it holds that:

- $\iota \leq \iota$: basic types like integers and strings are subtypes of themselves.
- $\tau_1 \leq \tau'_1, \dots, \tau_n \leq \tau'_n \Rightarrow \{a_1 : \tau_1, \dots, a_{n+m} : \tau_{n+m}\} \leq \{a_1 : \tau'_1, \dots, a_n : \tau'_n\}$: a record type $\tau = \{a_1 : \tau_1, \dots, a_{n+m} : \tau_{n+m}\}$ is called a subtype of another record type $\tau' = \{a_1 : \tau'_1, \dots, a_n : \tau'_n\}$ if it has all the typed labels of τ', and possibly m more, and the common labels are also in a subtyping relation (pair-wise). τ' is respectively called a super-type of τ.

This basic definition is used at the basis for the application of type theory to the service records we defined in Chapter 4. A service record (either an element or a relationship) is a tuple (and therefore a type of set) containing typed labels and their values. We can, and we do apply the same principles for subtyping the records of a service description as the criterion for compatibility between them.

On a relevant note, while Definition 8 covers the syntactic aspects of objects, it is not sufficient for comparing objects based on their behavior. For that reason a number of important works in the '90s, like [145], [125], [162], extended the notion of subtyping to the behavioral semantics of objects using pre- and post-conditions. The subtype relation in that context ensures that the subtype objects preserves the behavior of the supertype objects. With respect to the behavioral layer of service representation, it is feasible to adapt these theories in our model.

Furthermore, a more appropriate definition of subtyping for behavioral protocols (message exchanges) is provided in [144]; since the formalization of the behavioral layer in Chapter 4 was done on the basis of that work it is possible to reuse their definition for our purposes. In addition, in [153], we defined subtyping for the non-functional layer of service representation. The following sections are going to elaborate further on the definitions of subtyping for the structural, behavioral and non-functional records.

For the presentation of our approach on type theory we use the same logical organization as in Chapter 4. In particular, we start by defining subtyping for the structural layer of the ASD descriptions which acts as the foundation of the ASD model. Then we extend the definitions appropriately to the behavioral and non-functional aspects of services.

6.2.2 Structural Subtyping

As a first step for defining a type theory of ASD records we start by modifying Definition 8 to fit the more specific definition of what constitutes a record in the ASD model. More specifically, by their definition, each element and relationship are *types* themselves. We can therefore compare two elements or relationships by extending the subtyping relation as follows:

Definition 9
(Structural) Subtyping of elements and relationships

1. For $e = (name, att_1, \ldots, att_k, pr_1, \ldots, pr_l)$ and $e' = (name', att'_1, \ldots, att'_m, pr'_1, \ldots, pr'_n)$, we define the subtype relation between e and e' as:

$$\begin{aligned} e \leq e' \Leftrightarrow \quad & name \equiv name' \wedge \\ & k > m, att_i \leq att'_i, 1 \leq i \leq m \wedge \\ & l > n, pr_j \leq pr'_j, 1 \leq j \leq l \end{aligned}$$

   that is, they have the same (or equivalent – more on that later) name identifier, and e' has less attributes and properties than e, but the ones it has are more generic (super-types) of the respective attributes and properties of e. By definition it holds that $(e = \emptyset) \leq e'$.

2. For $r(e_s, e_t) = (name_s, name_t, rel, mul)$ and $r(e'_s, e'_t) = (name'_s, name'_t, rel', mul')$ we define the subtype relation between r and r' as:

$$r(e_s, e_t) \leq r(e'_s, e'_t) \Leftrightarrow e_s \leq e'_s \wedge e_t \leq e'_t \wedge rel = rel' \wedge mul \subseteq mul'$$

   that is, the elements e'_s, e'_t participating in the (new) relationship are super-types of e_s, e_t respectively (and therefore $name_s \equiv name'_s, name_t \equiv name'_t$) and the multiplicity domain of the relationship is a super-set of the respective one in the old relationship. We assume by definition that

$$(r(e_s, e_t) = \emptyset) \leq r(e'_s, e'_t) \Leftrightarrow \begin{cases} e_s \neq \emptyset \wedge mul' = [0, N'], \ N' \geq 1 \\ e_s = \emptyset \wedge e_t = \emptyset \end{cases}$$

   (either an optional relationship is added to an existing element or a relationship is added to a new element).

Corollary: The subtyping relation is by its definition a partial order (that is, it is reflexive, transitive and antisymmetric).

In the general case, the equivalence $name \equiv name'$ can be interpreted as synonymy, hyponymy or another similar semantic relationship. For the purposes of this work we do not consider the semantics of each record and equivalence is thus reduced to equality. The two elements should therefore have the same name identifier.

With respect to the attributes att_i and the properties in the property domain `DataType` of `Information Type` it holds that

$$int \leq double \leq \cdots \leq string \leq document$$

It also holds that

$$one - way \leq request - response \wedge \ notification \leq solicit - response$$

for the properties of the `MessagePattern` property domain in the `Operation` concept. This allows us to modify not only the message payload but also the interaction protocol of the service operations under certain conditions that we discuss in the following. Both of these options are not allowed by the preservation guidelines in Table 6.1. This comes as a natural and sound extension to current approaches to service compatibility. For the properties of the `MessageRole` property domain of `Message`s though, the subtyping relation holds only for equality (that is, $input \leq input, output \leq output$ and $fault \leq fault$)[2]. This means that the role of a message can not be changed without breaking compatibility.

```
<xsd:complexType name="PODocument">
 <xsd:sequence>
  <xsd:element name="OrderInfo" type="xsd:string"/>
  <xsd:element name="DeliveryInfo" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Listing 6.2: PopService WSDL – Change Scenario I

Consider for example the new version for the Change Scenario I of the PopService in Listing 3.3 and repeated in Listing 6.2 for convenience. The scenario requires the delivery information to be obligatorily submitted together with the purchase order (instead of optionally as in the previous version in Listing 3.1, as indicated by the `minOccurs="0"` attribute value). The $r(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, s, [0, 1])$ relationship (as defined in Section 4.2.1) is therefore replaced in the service description $\mathcal{S}'$ of the service by the relationship $r'(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, s, [1, 1])$. From Definition 9 it holds that $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$ since

[2]This also holds for all the other property domains in Fig. 4.1, except if explicitly stated otherwise in the discussion that follows.

- $e_{pod} \leq e_{pod} \ \wedge \ e_{di} \leq e_{di}$: e_{pod}, e_{di} are unchanged and by the reflexive property of subtyping they are subtypes of themselves,
- $rel' = rel = s$: the type of the relationship is also unchanged, and
- $e_{pod} \neq \emptyset \wedge mul' \subseteq mul$, since $[1,1] \subseteq [0,1]$.

$r'(e_{pod}, e_{di})$ is therefore a subtype of $r(e_{pod}, e_{di})$. This should be expected: an optional data type in the message schema is more generic than the same message schema with the data type as mandatory.

6.2.3 Behavioral Subtyping

In Chapter 4 we explained that we rely on the formalization of behavioral contracts (based in turn on the CCS calculus) provided by [143] and [144] for the description of the behavioral layer. The main reason for selecting this notation for the behavioral records is that it comes with a definition of *behavioral subcontract relation* $\preceq$ that checks the compatibility of two behavioral contracts. A behavioral contract σ is a subcontract of contract σ' iff σ manifests less interacting capabilities than σ'. For example, it holds that $\overline{a} \oplus \overline{b} \preceq \overline{a}$ (where $\oplus$ signifies the internal choice operator) since every client that can interact successfully with a service that chooses when to provide a or b does also with one that offers systematically a[3].

Applying the subtyping relation and checking for compatibility between versions of records in the behavioral layer is therefore reduced to mapping them to the respective behavioral contracts and applying the behavioral subcontracting relation $\preceq$ between them. This is achieved by overloading the semantics of the subtyping relation for `Protocol` elements and adding the following condition in Definition 9:

Definition 10

`Protocol` **Subtyping**

Any `Protocol` element $\mathtt{e}_{\mathtt{prt}}$ is a subtype of another `Protocol` element $\mathtt{e}'_{\mathtt{prt}}$ iff their behavioral contracts are in a subcontract relation, that is:

$$\mathtt{e}_{\mathtt{prt}} \leq \mathtt{e}'_{\mathtt{prt}} \Leftrightarrow \sigma(\mathtt{e}_{\mathtt{prt}}) \preceq \sigma(\mathtt{e}'_{\mathtt{prt}})$$

The addition of an option of *synchronous communication* mode to the input of PopService initiated by Change Scenario II in Chapter 3 for example results in a protocol that is a supertype of the initial protocol of the service. In Chapter 4 we showed that the BPEL

[3]The full formalization, together with the construction of the proofs for what constitutes more or less interacting capabilities is presented at length in [144]. For the purposes of this discussion we rely on the intuitive definition of the term.

description of the service as depicted in Listing 3.2 is mapped to the ASD records:

$$
\begin{aligned}
e_{sequence} &= (sequence) \\
e_{ReceivePO} &= (ReceivePO, receive) \\
e_{SubmitPOAck} &= (SubmitPOAck, invoke) \\
r(e_{sequence}, e_{ReceivePO}) &= (sequence, ReceivePO, follows, [1,1]) \\
r(e_{sequence}, e_{SubmitPOAck}) &= (sequence, SubmitPOAck, follows, [1,1])
\end{aligned}
$$

that are equivalent to the behavioral contract $\sigma(e_{sequence}) = a_{ReceivePO}.\overline{a_{SubmitPOAck}}$.

```
<pick>
 <onMessage partnerLink="Client" operation="receivePO"
  portType="ns:POPServicePortType" variable="PO">
  <sequence>
   ...
   <invoke name="SubmitPOAck" partnerLink="Client"
    operation="receivePOCallBack" portType="ns:POPServiceCallBackPortType"
    inputVariable="POAck"/>
   </sequence>
 </onMessage>
 <onMessage partnerLink="Client2" operation="receivePOSync"
  portType="ns:POPServicePortType2" variable="PO">
  <sequence>
   ...
   <reply name="ReplyPOAck" partnerLink="Client2"
    operation="receivePOSync" portType="ns:POPServicePortType2"
    variable="POAck"/>
  </sequence>
 </onMessage>
</pick>
```

Listing 6.3: PopService BPEL – Change Scenario II

The BPEL description of the PopService in Change Scenario II, partially repeated

in Listing 6.3 for convenience, is mapped to the ASD records:

$$
\begin{aligned}
e'_{pick} &= (pick) \\
e'_{seq_1} &= (seq_1) \\
e_{ReceivePO} &= (ReceivePO, receive) \\
e_{SubmitPOAck} &= (SubmitPOAck, invoke) \\
r'(e'_{seq_1}, e_{ReceivePO}) &= (seq_1, ReceivePO, follows, [1,1]) \\
r'(e'_{seq_1}, e_{SubmitPOAck}) &= (seq_1, SubmitPOAck, follows, [1,1]) \\
e'_{seq_2} &= (seq_2) \\
e'_{ReceivePOSync} &= (ReceivePOSync, receive) \\
e'_{ReplyPOAck} &= (ReplyPOAck, reply) \\
r'(e_{seq_2}, e_{ReceivePO}) &= (seq_2, ReceivePO, follows, [1,1]) \\
r'(e_{seq_2}, e_{SubmitPOAck}) &= (seq_2, SubmitPOAck, follows, [1,1]) \\
r'(e'_{pick}, e'_{seq_1}) &= (pick, seq_1, eChoice, [1,1]) \\
r'(e'_{pick}, e'_{seq_2}) &= (pick, seq_2, eChoice, [1,1])
\end{aligned}
$$

The equivalent expression in behavioral contracts is

$$\sigma(e'_{pick}) = (a_{ReceivePO}.\overline{a_{SubmitPOAck}}) + (a_{ReceivePOSync}.\overline{a_{ReplyPOAck}})$$

from which it can be seen that $\sigma(e_{sequence}) \preceq \sigma(e'_{pick})$ since $\sigma(e'_{pick})$ contains $\sigma(e_{sequence})$ and allows for further interactions. Therefore, and according to Definition 10 we can conclude that $e_{sequence} \leq e'_{pick}$. This means that a client that works with protocol $e_{sequence}$ can also work without a problem with protocol e'_{pick}.

Reasoning on the `Operation Conditions` and `Constraint` elements and their relationships, in the way we presented them in Chapter 4, is sufficiently covered by Definition 9. Adding new `Constraints` $e'_{con_i} = (con_i, expression_i, true)$ and $e'_{con_j} = (con_j, expression_j, true)$ to an existing `Operation Conditions` element $e_{opcon} = (opcon, pre-)$ for example, creates additional relationships $r'(e_{opcon}, e'_{con_i}) = (opcon, con_i, c, [1,1])$ and $r'(e_{opcon}, e'_{con_j}) = (opcon, con_j, c, [1,1])$ for which we know that $(r(e_{opcon}, e_{con_i}) = \emptyset) \not\leq r'(e_{opcon}, e'_{con_i})$ and $(r(e_{opcon}, e_{con_j}) = \emptyset) \not\leq r'(e_{opcon}, e'_{con_j})$ since $e_{opcon} \neq \emptyset \wedge [1,1] \neq [0,N], N \geq 1$.

6.2.4 Non-functional Subtyping

Extending the subtyping relation as defined in Definition 9 in the model of description of QoS dimensions we assumed in Chapter 4 requires two things: providing operators for ordering the value ranges for each assertion element with respect to how general/specific they are, and handling the special semantics of the assertion sets that combine assertions using disjunctions and conjunctions. For the former we base the ordering of assertions on the nature of their dimension (i.e. whether it is monotonic or antitonic) and we use the

relations already defined in Allen's Interval Algebra [163] for relatively positioning intervals (here value ranges) on a dimension [153]. For the latter we use the simple observation that an assertion set with more conjunctions is more restrictive (i.e. more specific) than one with less, while the reverse is true for disjunctions.

More specifically, and given the fact that any `Assertion` element is a tuple

$$\mathsf{e}_{\mathtt{asrt}} := (name, dimension, dimtype : dimensionType, value, role : Intention)$$

we define the following relations between assertion *value*s:

$=$ *value is equal* to length and overlaps totally with *value*′,

s *value starts* together with *value*′ but finishes before it,

f *value* starts after *value*′ but it *finishes* together with it,

m *value meets value*′ at its finishing point (*value* finishes when *value*′ starts),

o *value* partially *overlaps* with *value*′ – having started before *value*′,

$<$ *value takes place before value*′.

The inversions $\mathsf{si}, \mathsf{fi}, \mathsf{mi}, \mathsf{oi}, >$ signify that the roles of *value* and *value*′ are reversed (*value*>*value*′ e.g. means that *value*′ takes places before *value*, etc.) For value ranges $[7.5, 15]$ and $[15, 30]$ for example it holds that $[7.5, 15]$ m $[15, 30]$ and $[15, 30]$ mi $[7.5, 15]$, while for value ranges $[81, 100]$ and $[90, 100]$ that $[90, 100]$ f $[81, 100]$ and $[81, 100]$ fi $[90, 100]$. Fig. 6.2 illustrates the relevant positioning of the values for the relations between v and v'. Based on this relevant position we can start building the subtyping relation for `Assertion` elements: having two value ranges on e.g. a monotonic dimension, the one that is further on the "right" part of the axis in Fig. 6.2 (towards the larger values) can (potentially) replace the other, and therefore it can be considered a super-type of the latter.

In order to demonstrate this we consider the case of two value ranges for availability, with $v = [80, 100]$ (availability more than 80% on average) and $v' = [90, 100]$ (more than 90%); from above, it holds that v fi v'. The basic observation here is that a service consumer that can accept availability in the $v = [80, 100]$ range can also accept availability in the $v' = [90, 100]$ range since it does not affect his assumptions about the QoS characteristics of the service. Following the basic idea behind subtyping as described in Section 6.2.1, any value range v' with a maximum of 100 and a minimum of more than 80 can be considered a super-type of v in this manner. Reasoning on a similar fashion we can conclude that for monotonic dimensions it holds that: $v \leq v' \Leftrightarrow value$ op $value', \mathsf{op} \in \{=, <, \mathsf{s}, \mathsf{fi}, \mathsf{m}, \mathsf{o}\}$.

The inverse reasoning can be applied for antitonic dimensions: super-types are positioned further on the "left" side of the axis in Fig. 6.2, or more formally: $v \leq v' \Leftrightarrow$ $value$ op $value', \mathsf{op} \in \{=, >, \mathsf{f}, \mathsf{si}, \mathsf{mi}, \mathsf{oi}\}$.

These observations are summarized in Definition 11 which rules the subtyping between `Assertion` elements:

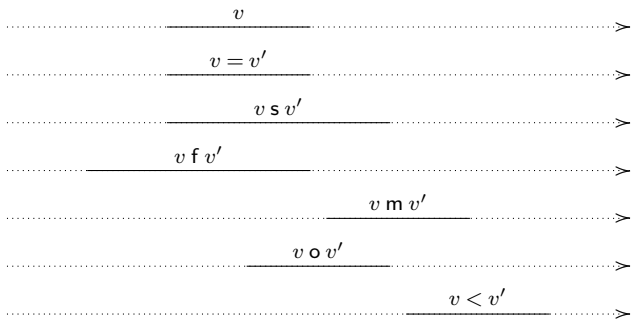


Figure 6.2: QoS values relations

Definition 11

`Assertion` **Subtyping**

An `Assertion` element $\mathsf{e}_{\mathtt{assert}} = (name, dimension, dimtype, value, role)$ is a subtype of another `Assertion` element

$$\mathsf{e}'_{\mathtt{assert}} = (name', dimension', dimtype', value', role')$$

iff:

$$\mathsf{e}_{\mathtt{assert}} \leq \mathsf{e}'_{\mathtt{assert}} \;\Leftrightarrow\; name \equiv name' \wedge dimension = dimension' \wedge dimtype = dimtype' \wedge \begin{cases} value \leq value' \wedge role = role' = promise \\ value' \leq value \wedge role = role' = obligation \end{cases}$$

with

$$value \leq value' \Leftrightarrow value \,\mathsf{op}\, value', \begin{cases} \mathsf{op} \in \{=, <, \mathsf{s}, \mathsf{fi}, \mathsf{m}, \mathsf{o}\} & \text{(monotonic dimensions)} \\ \mathsf{op} \in \{=, >, \mathsf{f}, \mathsf{si}, \mathsf{mi}, \mathsf{oi}\} & \text{(antitonic dimensions)} \end{cases}$$

As in the case of Definition 9, the two elements are connected by the subtyping relation iff they share the same name, dimension and dimension type. The properties *promise* and *obligation* in the `Intention` domain, as defined in Chapter 4, drive the interpretation of the value range subtyping. In the case that the assertions are offered as *promise*s to the service consumers then the subtyping of the values follows the reasoning discussed above – otherwise the reasoning is inversed.

Consider for example the non-functional description of the PopService as discussed in Chapter 4. The ASD records for the non-functional characteristics of the service are:

$$
\begin{aligned}
e_{assert_1} &= (assert_1, availability, monotonic, [80, 95], promise) \\
e_{assert_2} &= (assert_2, latency, antitonic, [15, 30], promise) \\
e_{assert_3} &= (assert_3, reliability, monotonic, [90, 100], promise) \\
e_{aset_1} &= (aset_1) \\
e_{pfl_1} &= (pfl_1) \\
r(e_{aset_1}, e_{assert_1}) &= (aset_1, assert_1, AND, [1, 1]) \\
r(e_{aset_1}, e_{assert_2}) &= (aset_1, assert_2, AND, [1, 1]) \\
r(e_{aset_1}, e_{assert_3}) &= (aset_1, assert_3, AND, [1, 1]) \\
r(e_{pfl_1}, e_{aset_1}) &= (pfl_1, aset_1, OR, [1, 1])
\end{aligned}
$$

Assertion elements e_{assert_1} - e_{assert_3} are QoS characteristics that the service guarantees to uphold when invoked, and as such they have the *promise* property. The changes to the QoS profile of the service introduced by Change Scenario I result in the following records:

$$
\begin{aligned}
e_{assert_1} &= (assert_1, availability, monotonic, [80, 95], promise) \\
e'_{assert_2} &= (assert_2, latency, antitonic, [7.5, 15], promise) \\
e'_{assert_3} &= (assert_3, reliability, monotonic, [81, 100], promise) \\
e'_{aset_1} &= (aset_1) \\
e'_{pfl_1} &= (pfl_1) \\
r(e'_{aset_1}, e_{assert_1}) &= (aset_1, assert_1, AND, [1, 1]) \\
r'(e'_{aset_1}, e'_{assert_2}) &= (aset_1, assert_2, AND, [1, 1]) \\
r'(e'_{aset_1}, e'_{assert_3}) &= (aset_1, assert_3, AND, [1, 1]) \\
r'(e'_{pfl_1}, e'_{aset_1}) &= (pfl_1, aset_1, OR, [1, 1])
\end{aligned}
$$

By their definition, latency is an antitonic dimension (lower values are better) and reliability is monotonic (the closer to 100%, the better). From these facts we can deduct from Definition 11 that $e_{assert_2} \leq e'_{assert_2}$ and $e'_{assert_3} \leq e_{assert_3}$ (with e_{assert_1} remaining unchanged) since:

1. $assert_2 = assert_2 \wedge latency = latency \wedge antitonic = antitonic \wedge promise = promise \wedge ([15, 30] \text{ mi } [7.5, 15] \Rightarrow [15, 30] \leq [7.5, 15]) \Rightarrow e_{assert_2} \leq e'_{assert_2}$
2. $assert_3 = assert_3 \wedge reliability = reliability \wedge monotonic = monotonic \wedge promise = promise \wedge ([81, 100] \text{ fi } [90, 100] \Rightarrow [81, 100] \leq [90, 100]) \Rightarrow e'_{assert_3} \leq e_{assert_3}$

In case that these assertions had the *obligation* property, signifying that they are expected to be fulfilled by an external party (requiring in this case from a service that is being consumed to offer the defined value ranges for availability, latency and reliability) then by Definition 11 we can see that the relations are inversed: $e'_{assert_2} \leq e_{assert_2}$ and

$e_{assert_3} \leq e'_{assert_3}$. This reflects the fact that the service can always lower its expectations from its environment (its consumers and the services it consumes) while it has to offer better QoS to its consumers.

Definition 11 supplements Definition 9 with the necessary constructs for comparing `Assertion` elements. In order to be able to reason on the organization of `Assertion`s into `Assertion Set`s, and `Assertion Set`s into `Profile`s using the `lType` logical relationship we need to further supplement Definition 9 accordingly:

Definition 12

`Assertion Set` and `Profile` Subtyping

An element $\mathtt{e}$, either an `Assertion Set` or a `Profile` element, is a subtype of another element $\mathtt{e}'$ (of the same concept) iff:

$$\mathtt{e} \leq \mathtt{e}' \Leftrightarrow \begin{cases} \forall \mathtt{e}_i \in \mathcal{S}, r(\mathtt{e}, \mathtt{e_i}, OR, mul) \in \mathcal{S}, \exists \mathtt{e}'_\mathtt{i} \in \mathcal{S}' / \\ r'(\mathtt{e}', \mathtt{e}'_\mathtt{i}, OR, mul') \in \mathcal{S}' \wedge \mathtt{e_i} \leq \mathtt{e}'_\mathtt{i} \wedge mul \subseteq mul' \\ \\ \forall \mathtt{e}'_\mathtt{i} \in \mathcal{S}', r'(\mathtt{e}', \mathtt{e}'_\mathtt{i}, AND, mul') \in \mathcal{S}', \exists \mathtt{e_i} \in \mathcal{S} / \\ r(\mathtt{e}, \mathtt{e_i}, AND, mul) \in \mathcal{S} \wedge \mathtt{e_i} \leq \mathtt{e}'_\mathtt{i} \wedge mul \subseteq mul' \end{cases}$$

Definition 12 combines the logic behind Definitions 9 and 10 in a recursive fashion. Any `Assertion Set` $\mathtt{e}_{\mathtt{assert}}$ is a subtype of another `Assertion Set` element $\mathtt{e}'_{\mathtt{assert}}$ iff its `Assertion` elements are subtypes of the respective `Assertion` elements of $\mathtt{e}'_{\mathtt{assert}}$ by Definition 11. The difference between the two legs is in the nature of the AND and OR `lType` relationships. The first leg of Definition 12 allows for more OR statements to be added – and therefore more options to the respective assertion set. The second leg on the other hand allows for AND statements to be removed – making the assertion set more generic by providing less restrictions in the form of QoS characteristics to be fulfilled. When applied to `Profile` elements, Definition 12 starts examining each `Assertion Set` under the `Profile` in a similar fashion; more options (OR relationships), where each option is also more general than before, denotes a super-type.

For the PopService for example we know from above that $e_{assert_1} \leq e'_{assert_1}$ (due to the reflexive property of the relation), $e_{assert_2} \leq e'_{assert_2}$ and $e'_{assert_3} \leq e_{assert_3}$. From this we conclude that $e_{aset_1} \not\leq e'_{aset_1}$ due to $e'_{asssert_3}$ and therefore also $e_{pfl_1} \not\leq e'_{pfl_1}$: by offering less reliability to its consumers, PopService is asking them to accept less QoS than it had initially promised and therefore breaks their expectations. If $e_{asssert_3}$ had remained unaffected or removed completely (and just decreased its latency, as expressed by element e'_{assert_2}) for the PopService ASD then we could conclude that $e_{aset_1} \leq e'_{aset_1}$ from the second leg of the definition and therefore $e_{pfl_1} \leq e'_{pfl_1}$.

6.3 Reasoning on Service Evolution

Having established a type theory for all the layers of an ASD, it becomes possible to use the subtyping relation of ASD records to check for the compatibility of service versions.

Reasoning about this decision is quite straightforward: by combining Definitions 7 and 9 (as extended for each layer by Definitions 10 and 12) we can check whether both cases of compatibility are satisfied using the definition of subtyping for ASDs. The following sections discuss how this can be achieved, starting by classifying changes based on whether they respect Definition 7 or not.

6.3.1 T-shaped Changes

In Section 5.3 we defined a *change set* $\Delta\mathcal{S}$ as a set of change primitives (fundamental modifications) that when applied to an ASD $\mathcal{S}$ it results into a new version $\mathcal{S}'$. We classify the change sets with respect to compatibility:

Definition 13

T-shaped changes

A change set $\Delta\mathcal{S}$ is called *T-shaped*, and we write $\Delta\mathcal{S} \in \mathbb{T}$ where $\mathbb{T}$ is the set of all possible T-shaped changes, if and only if, when $\Delta\mathcal{S}$ is applied to a service description $\mathcal{S}$ it results into a fully compatible with $\mathcal{S}$ service description $\mathcal{S}' = \mathcal{S} \circ \Delta S$, that is, $\mathcal{S} <_c \mathcal{S}'$.

Corollary: It holds by definition: $\forall\Delta\mathcal{S} \in \mathbb{T} : \mathcal{S} <_c \mathcal{S} \circ \Delta\mathcal{S}$ and $\forall\mathcal{S}', \mathcal{S} <_c \mathcal{S}' : \Delta\mathcal{S} \in \mathbb{T}$.

The term "T-shaped change" refers to the relation between the two aspects of compatibility as illustrated in Fig. 6.1. As long as a change set $\Delta\mathcal{S}$ results in a horizontally or vertically compatible (or both) version of a service, then it belongs to the set $\mathbb{T}$ of all possible T-shaped changes. *Constraining the evolution of services is therefore reduced to deciding whether* $\Delta\mathcal{S} \in \mathbb{T}$.

Reasoning on a change set is performed in three steps:

1. Calculation of the new version of the service by applying the change set to it $\mathcal{S}' = \mathcal{S} \circ \Delta\mathcal{S}$.
2. Distribution of the elements of $\mathcal{S}$ and $\mathcal{S}'$ in subsets $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$, and $\mathcal{S}'_{pro}$ and $\mathcal{S}'_{req}$, respectively (following Definition 6).
3. Use of Definition 7 to check whether $\Delta\mathcal{S} \in \mathbb{T}$ or not.

The first step is an application of the definition of the change sets and can be performed trivially. For the second step, the creation of the *pro* and *req* subsets we initially select all elements of input or output type in Fig. 4.1, starting with elements like `Message`s. Then, by taking advantage of the relationships between elements in Fig. 4.1, we propagate this property to all elements that are connected to them, following the direction of the arrow of the relationship. Then, we "mark" both the relationship and the connected element with the same type (input or output) and we continue this process until there are no more relationships to traverse.

	$\mathcal{S}_{pro}$	$\mathcal{S}_{req}$
Structural layer	`Message` elements with property `role=output` or `fault` & all `Information Type` elements that are related to them	`Message` elements with property value `role=input` & all `Information Type` elements that are related to them
Behavioral layer	`Activity` elements with property `act=invoke` or `act=reply` `Operation Conditions` elements with property `role=post-` & all `Constraint` elements that are related to them	`Activity` elements with property `act=receive` `Operation Conditions` elements with property `role=pre-` & all `Constraint` elements that are related to them

Table 6.2: Distribution of ASD elements $\mathcal{S}_{pro}$ and $\mathcal{S}_{req}$ sets

Table 6.2 summarizes the elements that are in the $\mathcal{S}_{pro}$ or $\mathcal{S}_{req}$ subset. A record can be in both subsets. An `Information Type` element for example can be used as a part in both an input and an output message of the service. This does not affect the rest of the reasoning: the element (and its relationships) will be handled as two distinct elements depending on whether we are reasoning on the $\mathcal{S}_{pro}$ or the $\mathcal{S}_{req}$ subset. For the last step, determining if a change set is T-shaped, the *Compatibility Checking Algorithm* (Algorithm 1) is used.

The first two steps of Algorithm 1 correspond to the two legs of Definition 7: the first one (lines 1 to 6) checks for the the covariance of input and the second one (lines 7 to 12) for the contravariance of output. For that purpose they use the definition of subtyping for records in different layers of the ASD as discussed in the previous section. The third step (lines 13 to 18) and fourth step (lines 19 to 24) ensure that the behavioral and non-functional aspect of ASDs in terms of `Protocol` and `Profile` elements also respect compatibility. It is necessary to perform these extra check since:

1. `Protocol` elements, $\mathtt{e}_{\mathtt{prt}}$ elements are neither in the $\mathcal{S}_{pro}$, nor in the $\mathcal{S}_{req}$ subset. This omission is by design: protocols include both input and output type elements, and therefore applying Definition 7 to them is not directly possible. Nevertheless, protocol subtyping as described by Definition 10 provides us with the means to check whether the protocols contained in $\mathcal{S}$ are also covered by the protocols of $\mathcal{S}'$.

2. `Profile` elements $\mathtt{e}_{\mathtt{pfl}}$ – as all other elements and relationships in the non-functional layer – are not distributed in the $\mathcal{S}_{pro}$ or $\mathcal{S}_{req}$ subsets. This is due to the fact that the QoS characteristics promised and expected by the service, as encoded in the $\mathtt{e}_{\mathtt{pfl}}$ elements, are defined on combinations of input and output (e.g. latency). For this reason they have to treated separately, using Definition 12 (and therefore also Definition 11) to allow the replacement of `Profile` elements only by more "general"

Algorithm 1 Compatibility Checking Algorithm

1: {*** Step 1 – $\mathcal{S}_{req}$ records ***}
2: **if** $\forall s' \in \mathcal{S}'_{req}, \exists s \in \mathcal{S}_{req}, s \leq s'$ **then**
3: continue to next step;
4: **else**
5: **return** *No*;
6: **end if**
7: {*** Step 2 – $\mathcal{S}_{pro}$ records ***}
8: **if** $\forall s \in \mathcal{S}_{pro}, \exists s' \in \mathcal{S}'_{pro}, s' \leq s$ **then**
9: continue to next step;
10: **else**
11: **return** *No*;
12: **end if**
13: {*** Step 3 – (behavioral) protocols ***}
14: **if** $\forall \mathtt{e}_{\mathtt{prt}} \in \mathcal{S}, \exists \mathtt{e}'_{\mathtt{prt}} \in \mathcal{S}', \mathtt{e}_{\mathtt{prt}} \leq \mathtt{e}'_{\mathtt{prt}}$ **then**
15: **return** continue to next step;
16: **else**
17: **return** *No*;
18: **end if**
19: {*** Step 4 – (non-functional) profiles ***}
20: **if** $\forall \mathtt{e}_{\mathtt{pfl}} \in \mathcal{S}, \exists \mathtt{e}'_{\mathtt{pfl}} \in \mathcal{S}', \mathtt{e}_{\mathtt{pfl}} \leq \mathtt{e}'_{\mathtt{pfl}}$ **then**
21: **return** *Yes*;
22: **else**
23: **return** *No*;
24: **end if**

characteristics (that is, their super-types).

The following section uses the change scenarios defined in Chapter 3 in order to better illustrate how reasoning on T-shaped changes is performed using Algorithm 1.

6.3.2 T-shaped Changes: Change Scenarios I-III

Change Scenario I

The first of the change scenarios results in changes in both the structural and non-functional layers of PopService. More specifically and as discussed throughout the previous sections,

$\mathcal{S}'$ differs from $\mathcal{S}$ by the following records:

$$
\begin{aligned}
r'(e_{pod}, e_{di}) &= (PODocument, DeliveryInfo, s, [1,1]) \text{ (in the structural layer)} \\
e'_{aset_1} &= (aset_1) \\
e'_{pfl_1} &= (pfl_1) \\
r(e'_{aset_1}, e_{assert_1}) &= (aset_1, assert_1, AND, [1,1]) \\
r'(e'_{aset_1}, e'_{assert_2}) &= (aset_1, assert_2, AND, [1,1]) \\
r'(e'_{aset_1}, e'_{assert_3}) &= (aset_1, assert_3, AND, [1,1]) \\
r'(e'_{pfl_1}, e'_{aset_1}) &= (pfl_1, aset_1, OR, [1,1]) \text{ (in the non-functional layer)}
\end{aligned}
$$

for which we have already established in the previous that

- $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$, with $r(e_{pod}, e_{di}) \in \mathcal{S}_{req}$ and $r'(e_{pod}, e_{di}) \in \mathcal{S}'_{req}$
- $e_{pfl_1} \not\leq e'_{pfl_1}$ since $e'_{assert_3} \leq e_{assert_3} \Rightarrow e_{aset_1} \not\leq e'_{aset_1}$

By combining the above we get that the change set $\Delta\mathcal{S}_I$ required by the scenario is not T-shaped: both Steps 1 and 4 of the compatibility checking algorithm (Algorithm 1) are violated since $r'(e_{pod}, e_{di}) \leq r(e_{pod}, e_{di})$ and $e_{pfl_1} \not\leq e'_{pfl_1}$, respectively. In service versioning terms, this signifies the need for the creation of a major version of the service, requiring the consumers of PopService to adapt or migrate to the new version.

This scenario illustrates the case for shallow changes: by trying to minimize errors and improve the performance of the service, the service developers unintentionally generate additional development effort for the service consumers. While this cost may appear small, it is impossible to predict the actual impact of such a change for SBAs consuming the service if a re-engineering effort is required. Furthermore, it has to be considered that the creation of the new version of PopService must be accompanied by the execution of an appropriate decommissioning plan for the existing version to facilitate the transition to the new version (as discussed in Chapter 5). This plan comes with additional costs in communicating the change to the consumers and running two active versions of the service (and their supporting implementation) in parallel for the transitional period. The costs of implementing Change Scenario I therefore may outweigh its benefits and in this case it has to reconsidered.

Change Scenario II

This scenario has two major effects on the PopService: it changes its interaction protocol by replacing the simple sequence with a pick activity, and adds a new operation to the structural description to support the additional entry point. We already showed in Section 6.2.3 that with respect to the `Protocol` elements it holds that $e_{sequence} \leq e'_{pick}$ and thus passes Step 3 of the algorithm. Algorithm 1 therefore returns 'Yes' if it passes Steps 1, 2 and 4.

The addition of the `receivePOSync` operation to the WSDL of the service depicted in Listing 3.4 is mapped in ASD notation to the addition of element $e'_{recsync}$ to $\mathcal{S}$, together with its (structural) relationships $r'(e'_{recsync}, e_{msg})$ and $r'(e'_{recsync}, e_{msgack})$ to the existing `POMessage` and `POMessagesAck` messages, respectively. Furthermore, from Section 6.2.3, the elements $e'_{pick}, e'_{seq_1}, e'_{seq_2}, e'_{ReceivePOSync}, e'_{ReplyPOAck}$ and the respective relationships have to be added. In addition, the $e_{sequence}$ is removed and replaced by e_{pick}. For these elements it holds $e'_{recsync}, e'_{ReceivePOSync} \in \mathcal{S}'_{req}$ and $e'_{ReplyPOAck} \in \mathcal{S}'_{pro}$. Since according to Definition 9 it holds

- $\emptyset \leq e'_{recsync}$,
- $\emptyset \leq e'_{ReceivePOSync}$,
- $\emptyset \leq r'(e'_{recsync}, e_{msg})$,
- $\emptyset \leq r'(e'_{recsync}, e_{msgack})$, and
- $\emptyset \leq r'(e'_{ReceivePOSync}, e'_{recsync})$

and the rest of $\mathcal{S}'_{req}$ is unchanged, then Step 1 of Algorithm 1 passes to the next step. Since no existing element that belongs to $\mathcal{S}_{pro}$ was affected by the change set $\Delta\mathcal{S}_{II}$ then Step 2 also passes. Step 3 has been already confirmed to pass and since there was no change in the non-functional aspect of the service then Step 4 return 'Yes' and therefore $\Delta\mathcal{S}_{II} \in \mathbb{T}$. In other words, despite the major changes required to the service description, this scenario requires only a minor version of the service to be created.

In contrast to Change Scenario I, Scenario II is shallow. This means that the new version of PopService $\mathcal{S}'$ can be implemented and deployed by replacing the previous version *without any effect to existing consumers*. Both new (using the synchronous communication capability) and old (using the asynchronous one) consumers can interact with the service using the same service interfaces. No particular decommissioning plan is necessary, and no additional costs (further than the development of the new service) are required. Being able to reason that Change Scenario II is T-shaped therefore guarantees that the effort and impact of implementing the change is minimum.

Change Scenario III

It is trivial to show that $\Delta\mathcal{S}_{III}$, the addition of a time stamp to all incoming and outgoing messages defined by Change Scenario III is not T-shaped. For that purpose it suffices to show that there exists a record in $\mathcal{S}'_{req}$ or $\mathcal{S}'_{pro}$ that does not respect Definition 9.

The element denoting the `PODocument` in Listing 3.1 for example was defined in Chapter 4 as the element $e_{pod} = (PODocument, document)$ with relationships $r(e_{pod}, e_{oi}) = (PODocument, OrderInfo, s, [1, 1])$ for the `OrderInfo` part and $r(e_{pod}, e_{di}) = (PODocument, DeliveryInfo, s, [0, 1])$ for the `DeliveryInfo` part. The new element e'_{pod} differs from e_{pod} by having an additional relationship $r'(e'_{pod}, e'_{ts}) = (PODocument, TimeStamp, s, [1, 1])$ with new element $e'_{ts} = (TimeStamp, dateTime)$

representing the time stamp information. From Definition 9 we can conclude that $e'_{pod} \leq e_{pod}$. This means then that $\exists s' = e'_{pod} \in \mathcal{S}'_{req}$, $\nexists s \in \mathcal{S}_{req}, s \leq s'$ and Step 1 of Algorithm 1 fails for e'_{pod}; therefore $\Delta\mathcal{S}_{III} \notin \mathbb{T}$. Again, this calls for a major version of the service to be created and deployed.

As in the case of Change Scenario I, Change Scenario III therefore requires service consumers to migrate or adapt to the new version. Contrary to Scenario I however, in this case it is not possible to avoid the cost of such transition. The requirement for change and even the decommissioning period comes from "above". Both service provider and service consumers must share the costs of this transition. Change Scenario III is an example of a deep change: a regulatory modification leads to a series of changes across the service chain, with an unknown impact to its members.

6.4 Comparison with Existing Approaches

The proposed approach builds a theory for service compatibility and shows how the compatibility between different versions of the service can be preserved. In the following we qualitatively evaluate our proposal by comparing it with the guideline-driven preventive service approaches and examining its *novelty* and *relevance* to the service-specific aspects of SBAs.

6.4.1 Compatible Change Patterns

In order to compare our theory with the existing approaches we need to examine whether it is possible to (at the bare minimum) generate the T-shaped change sets that correspond to the backward compatibility preservation guidelines in Table 6.1. The results of this procedure are summarized in Table 6.3 which contains a number of *compatible patterns of change sets*. Some of these patterns correspond to the backward-compatibility preservation guidelines in Table 6.1. The rest of the patterns define guidelines that are not contained in Table 6.1 and are indicated by being written in italics (in the right-most column). Each pattern is accompanied by an explanation on the reasoning that leads to the T-shaped property[4].

In particular, $\Delta\mathcal{S}_{P1}$ corresponds to the guideline of adding optional message data types to input. As shown, $\Delta\mathcal{S}_{P1}$ is T-shaped irrespective of whether the data types (represented by an *it* element) are added to a message that belongs to the provided or required set – in either input or output. That is because if it is the former case, then it does not affect the reasoning on Step 1 of the compatibility checking algorithm; if it is the latter case, then due to the fact that an optional relationship (with minimum multiplicity 0) is a super-type of the "empty" relationship by definition, and given that the rest of $\mathcal{S}$ remains unaffected, then it also passes Step 2. $\Delta\mathcal{S}_{P1}$ is therefore more general than the corresponding guideline. $\Delta\mathcal{S}_{P2}$ is also T-shaped under all cases following a similar reasoning.

[4]In the table and the following discussion we will write *msg* denoting a `Message` element, and *it* for `Information Type` and *op* for `Operation` elements, respectively.

Pattern	**T-shaped Change**	**Guideline in Table 6.1**
$\Delta\mathcal{S}_{P1} = \{add(it', \mathcal{S}), add((msg_i, it'), \mathcal{S})\}, r(msg_i, it') = \{msg_i.name, it'.name, a, mul\}, mul = [0, N), N > 0\}$	Yes, if $msg_i \in \mathcal{S}_{pro}$ then there is no violation of covariance; if $msg_i \in \mathcal{S}_{req}$ then it holds by definition that $\emptyset \leq r(msg_i, it')$.	Add (Optional) Message Data Types
$\Delta\mathcal{S}_{P2} = \{add(op', \mathcal{S}), add((op', msg_i), \mathcal{S})\}$ or $\Delta\mathcal{S}_{P2} = \{add(op', \mathcal{S}), add(msg', \mathcal{S}), add((op', msg'), \mathcal{S}), \dots\}$	Yes, reasoning in a similar fashion as above.	Add (New) Operation
$\Delta\mathcal{S}_{P3} = \{del(op_i, \mathcal{S}), del((op_i, msg_{i,j}), \mathcal{S}), \dots\}$	No, if $\exists j, msg_{i,j} \in \mathcal{S}_{pro}$ due to covariance; Yes, otherwise (i.e. an one-way operation for example can be deprecated without an effect on the consumer - the service can just ignore the incoming message.)	(*Remove Operation*)
$\Delta\mathcal{S}_{P4} = \{mod(op_i, \mathcal{S})\}$ or $\Delta\mathcal{S}_{P4} = \{mod((op_i, msg_{i,j}), \mathcal{S})\}, r'(op_i, msg_{i,j}) = \{\dots, mul'\}\}$	Yes, if $mul \subseteq mul' \wedge msg_{i,j} \in \mathcal{S}_{req}$ (contravariance) or $mul' \subseteq mul \wedge msg_{i,j} \in \mathcal{S}_{pro}$ (covariance); No, otherwise.	(*Modify Operation*)
$\Delta\mathcal{S}_{P5} = \{mod(it_i, \mathcal{S})\}$ or $\Delta\mathcal{S}_{P5} = \{mod((it_i, it_{i,j}), \mathcal{S})\}, r'(it_i, it_{i,j}) = \{\dots, mul'\}\}$	Yes, if $mul \subseteq mul' \wedge it_i, it_{i,j} \in \mathcal{S}_{req}$ (contravariance) or $mul' \subseteq mul \wedge it_i, it_{i,j} \in \mathcal{S}_{pro}$ (covariance); No, otherwise.	(*Modify Message Data Types*)
$\Delta\mathcal{S}_{P6} = \{add(it', \mathcal{S}), add((msg_i, it'), \mathcal{S})\}, r(msg_i, it') = \{msg_i.name, it'.name, a, mul\}, mul = [M, N), 0 < M < N\}$	Yes, iff $msg_i \in \mathcal{S}_{pro}$ (covariance); No, for all other cases.	*Add Mandatory Data Types*
$\Delta\mathcal{S}_{P7} = \{del(it_i, \mathcal{S}), del((it_j, msg_{j,i}), \mathcal{S}), \dots\}$	Yes, iff $it_i \in \mathcal{S}_{req}$ (contravariance); No for all other cases.	*Remove Data Types*

Table 6.3: Patterns of Change Sets

$\Delta\mathcal{S}_{P3}$ on the other hand is T-shaped only if the deleted operation has only input messages and *under the assumption that these messages can be ignored without affecting either the producer or the consumer*. The respective guideline explicitly forbids this change set by being too conservative for the sake of safeness. Our approach shows that such a modification to a service would not necessarily break existing consumers. If the receipt of the message is part of a larger communication protocol though, then this change set may not be T-shaped due to the respective constraints on behavioral layer.

$\Delta\mathcal{S}_{P4}$ and $\Delta\mathcal{S}_{P5}$ work in a different manner. They allow making the input messages and their associated data types more flexible (by allowing a more general multiplicity domain in their relationship). This implies that the service can accept more incoming messages or a wider message payload than before. They also restrict the output messages accordingly. This means for example that the change from the service version in Listing 3.6 to the one in Listing 3.1 is T-shaped (since the multiplicity domain of the former is more general than that of the latter), but not the other way around.

$\Delta\mathcal{S}_{P6}$ and $\Delta\mathcal{S}_{P7}$ are not contained in Table 6.1 due to their specific nature: building on the same assumption as $\Delta\mathcal{S}_{P3}$, they accept as T-shaped the addition of a non-optional data type to an output message and the removal of a message data type for input messages. As with the other patterns, the reasoning is the same: as long as the consumer or the producer respectively can ignore the "additional" message payload then the compatibility is preserved. Further T-shaped change sets can be generated in a similar fashion.

The set of T-shaped change sets that can be produced by enumerating all possible change sets and checking them for compatibility is therefore a super-set of the guidelines-based one in Table 6.1. Enumerating all possible T-shaped change sets, even by starting with a simple meta-model as that of Fig. 4.1, is too lengthy of a process to be presented here and beats the purpose of this service compatibility theory. Nevertheless if necessary or desired, it is shown that this process is feasible.

6.4.2 Novelty

The proposed approach applies and extends a well tested and tried theory to software development practices to address service compatibility issues in service engineering. More specifically, it interweaves different aspects of type theory (structural, behavioral and non-functional) in a common framework for the description and consistent evolution of services. In that respect, it is broader and more powerful than the existing approaches on service evolution summarized in Table 5.1 that focus mainly on structural changes.

The independence from the specifics of a technology inherent in our approach allows for it to be applied to different service-enabling environments. Moving from WSDL 1.1 to WSDL 2.0 for example requires only the update of the ASD Meta-model, keeping the rest of the compatible service evolution model intact. In order to achieve this transition using the guidelines-based approaches we would have to redefine them, in order to accommodate the changes in the new specification.

Furthermore, the use of a common meta-model that brings together structural, behavioral and non-functional aspects of services allows for a uniform treatment of and efficient

reasoning about the compatibility of different service versions. Type theory-based approaches like [42], [43] and [44] have attempted a similar treatment for component-based systems. However, the lack of a meta-model for the description of components and the emphasis on the method signatures as the basic interaction mechanism with the component led to very fine-grained compatibility theories with limited applicability. Using a high-level, coarse-grained description model allowed us to build a more generic and efficient reasoning mechanism.

This reasoning can additionally be used in conjunction with alternative approaches for compatible service evolution like the semi-automatic generation of adapters – mediating software that resolves the mismatches between services and clients (as discussed in Chapter 2). In addition, the approach is extensible in that it allows for the proposed theoretical model to capture hitherto independent research threads which are relevant to service evolution such as contracting [131] (as we will discuss in the following chapter) and compliance checking [164] (which is outside the scope of this work).

The service compatibility theory and the compatible service evolution model presented in the above can serve as the foundation for developing sound and novel methods for designing and developing consistent service versions such as design techniques for change management as the ones mentioned in [112]. These methods are applicable to deep service changes and as such they are outside the scope of this work. Nevertheless, the work developed here can be perceived as the starting point for more complicated service evolution management solutions.

6.4.3 Relevance

The approach presented in this work is designed primarily with the field of service engineering and SBAs in mind. It achieves this by critically assessing, fusing together and extending select parts of existing theories (e.g. type theory) and techniques (e.g. software versioning) from diverse fields. The compatible service evolution model presented here relies on appealing properties of existing paradigms – such as component-based development – like information hiding, modularization and separation of concerns. It also extends those properties with features that are specific to services such as loose coupling, composition at the process level, asynchronous message-based invocation and coarse-grained interfaces that operate on a process level. The more representative SOA-specific points will be briefly described and contrasted with existing approaches like component-based development in the following:

1. *Type of Communication:* Evolving services in the compatible service evolution model use both synchronous and asynchronous communication to perform computations. While simple services can be developed using a request-response RPC-style synchronous behavior with fine-grain interactions, process-based services, i.e. composed services, require a more loosely-coupled asynchronous mode, which is typical of message-based systems. This can be contrasted with component-based development approaches, which are RPC-based.

2. *Type of Coupling:* Evolving services in our model make use of abstract message definitions to mediate their binding with respect to each other. This means that they focus on message definitions rather than method signatures which is the norm with component-based development approaches. This supports general-purpose message definitions such that the application code can independently handle the complexity of processing specific message instances that may change. This approach renders service interfaces reusable.

3. *Type of Interface:* The approach discussed in this work concentrates on coarse-grained interfaces between service providers and clients. By using this approach, the only assumption a service client makes is that the recipient will accept the message being sent. The client makes no assumptions about what will happen once the message is received. This is very specific to services. For instance, in the Automotive Purchase Order Processing scenario from Chapter 3, an inventory service would expose the inventory replenishment function and associated parameters. In contrast, a component-based development approach concentrates on object-level interfaces, as it will expose an entire inventory object with all its interfaces, as well as a replenishment object with all its interfaces. This requires the client to make many assumptions about the communication with the provider on a very low level.

4. *Type of Invocation:* The proposed approach deals with services that can be invoked under different categories, e.g. manufacturing or logistic services. Here the SBA may choose the most appropriate service on the basis of QoS by e.g. using parameters such as response times, throughput, availability and so on. This again is contrasted with component-based development approaches that focus on locating services by name.

In summary, the compatible evolution model ensures that service clients using a specific service that is upgraded in accordance with the preservation of compatibility do not experience disruptive changes. Otherwise service changes will most certainly result in severe application disruption, requiring radical modifications in the very fabric of the client services or the way that service-based applications using an upgraded service perform. In this way, service changes are always controlled, allowing services to evolve gracefully, ensure service stability, and handle structural, behavioral and non-functional variability.

6.5 Summary

Due to the overloading of the term compatibility in service (and not only) literature we started this chapter by first informally and then formally defining the term. For that purpose we integrated different definitions from programming languages, component-based systems and language producing theories into a concise service compatibility definition. We examined how service compatibility is supported by existing approaches in the field and we found them lacking. As a result we developed a service compatibility theory based on type theory to support the compatible evolution of services.

The developed theory provided us with the necessary conditions for ensuring the compatibility of service versions. Change sets that respect the compatibility of service versions are called T-shaped and are by definition shallow. We also showed how to reason on the evolution of services by the means of an algorithm that checks whether a change set is T-shaped. This reasoning is performed in a uniform way across all layers of the service (structural, behavioral and non-functional) as it has been demonstrated using the change scenarios introduced in Chapter 3. The effect of each scenario with respect to the effort required for implementing the change has also been discussed in relation to the shallow and deep nature of the change.

As a final step, the proposed approach was evaluated in a qualitative manner by comparing it with existing approaches. The comparative analysis performed showed that our approach is more general and fine-grained than similar proposals, having a theoretical foundation to rely on for producing its results instead of using best practices. We also showed that while conceptually similar approaches have been proposed before, the service oriented context that it is applied to makes it more suitable and efficient. Furthermore, some interesting extensions of the proposed theory were discussed. One of these extensions, the service contract formation and evolution, is discussed in the following chapter.

Chapter 7

Service Contracts

I watch the ripples change their size
But never leave the stream

David Bowie in his "actor" persona

Each thing is growing and decaying at the same time, only at different rates.

Balthasar Holz

The discussion in the previous chapter focused on the vertical compatibility aspect – that of the replaceability or substitutability (depending on the viewpoint adopted) of service versions. This chapter emphasizes the horizontal aspect, that of interoperability, while discussing the compatible evolution of services. More specifically, in Chapter 6 we discussed service compatibility as a pre-condition for shallow changes. In the following we show how we can expand the allowed changes to not necessarily compatible change sets (according to our previous definition), that, in addition to the T-shaped ones, ensure that the change to a service is shallow.

In order to achieve this goal we introduce the notion of *contracts* between service providers and service consumers. Contracts allow us to reason on the evolution of services in a horizontal, provider-to-consumer manner, whereas T-shaped changes reasoned in a vertical, provider-to-provider and consumer-to-consumer manner. Contractually-bound service evolution is as such better equipped to deal with interoperability issues than compatible service evolution, which focused on the replaceability/substitutability aspect. This, however occurs at the expense of additional reasoning, coupling, governance overhead and technical infrastructure.

The rest of this chapter briefly discusses service contracts and their life cycle. Using the example of a consumer for the Purchase Order Processing Service (PopService), we present how to form a contract between two interacting parties by re-using and extending tools we already developed in the previous chapters. We then show how contractually-bound service evolution can occur, and in which ways it can be more flexible than compatible service evolution. Furthermore, we discuss how even the contracts themselves can

evolve, enabling further the bounds of the possibilities for service evolution. We conclude this chapter by critically evaluating the proposed approach, discussing its advantages and disadvantages and close with a short summary of the chapter.

7.1 Service Contracts Life Cycle

Service contracts[1] are bilateral agreement between service providers and consumers that formalize the details of the provisioning of service (contents, protocols, delivery process, quality characteristics etc.) in a way that meets the mutual understandings and expectations of both parties [131], [132]. A service contract in this context is an intermediary between providers and consumers, expressed in the form of an ASD representation.

A service contract has a life cycle that runs in parallel with the service life cycle and consists of various phases from creation to decommissioning. For the purposes of this discussion we generalize the contract life cycle model developed in [153] for QoS contracts (also known as SLAs), shown in Fig. 7.1.

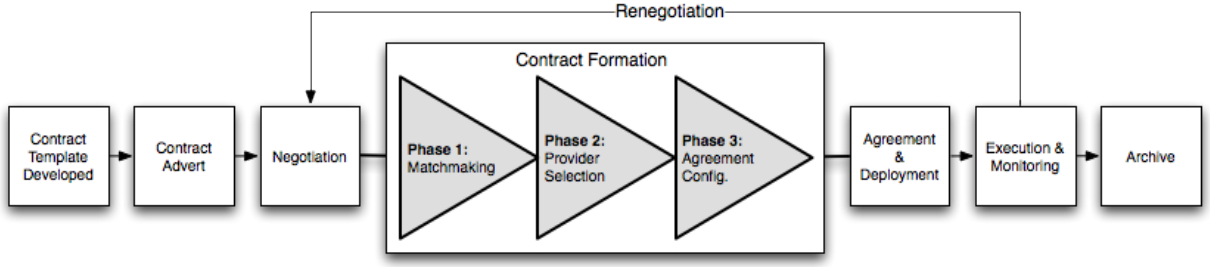


Figure 7.1: Contracts Life Cycle

The stages of the life of service contracts in this model are in summary:

- *Contract Template Development*: the blueprint of a contract (expressed for example as an ASD) is developed.
- *Contract Advertisement*: the blueprint is published to a service registry (if available), or otherwise simply deployed together with the service.
- *Negotiation*: interested service consumers enter into negotiation with the provider to define the characteristics included in the contract and acceptable values for them.
- *Contract Formation*: a contract is formed between the service provider and the interested service consumers.
- *Agreement & Deployment*: the contract is accepted from both parties and it is deployed on both parties and/or to an external contract broker.

[1]Not to be confused with the behavioral contracts defined by Castagna et al. [144] and used throughout the previous chapters. Even though we use their terminology for the subcontracting relation, their definition of a contract is essentially unilateral and for that purpose it was discussed in Chapter 4.

- *Execution & Monitoring*: the performance of the service is monitored either independently by the two parties, or externally by a third trusted party and the compliance to the contract terms is checked. Renegotiation of the contract terms is performed if major deviations are observed.
- *Archive*: the contract is decommissioned, signaling either the deprecation of the service, or its redesign and the subsequent restart of the contract life cycle.

Of particular interest for our purposes is the contract formation, that can be further decomposed in three phases:

1. *Matchmaking*: different service versions are checked against the requirements of the service consumer and only the most suitable ones are selected.
2. *Provider Selection*: from all suitable service versions, the most appropriate one with respect to the consumer's requirements is selected.
3. *Contract Configuration*: a contract is formed and finalized between the two parties.

Since our goal is to show how contracts can be formed and used as an intermediary for service evolution we focus on the matchmaking and configuration phases. Selection of the provider is handled implicitly through the matchmaking, since the contract formation model we develop in the following filters out all non-suitable versions in the matchmaking phase. Any version that comes out of this process can be used for contract configuration, if so required. The reader is referred to [130] for a wider discussion and alternative models for each of these stages.

For the purposes of showing how evolution can be facilitated through contracts we use the theory we developed in [131] that ties up with the service description and compatibility theories we discussed in the previous. For illustrative purposes we use the PopService defined in Chapter 3 and the Change Scenario I, that as we saw in the previous chapter, is not T-shaped. Since service contracts require two interacting parties we need to define a consumer for this service.

7.2 Interlude: A Consumer for the Purchase Order Processing Service

Based on the PopService we assume the existence of a PopClient, a dedicated SBA that uses PopService but operates under less strict QoS requirements. More specifically:

On the structural layer, PopClient uses Listing 3.1 but due to shipments to multiple delivery locations, the SBA designers chose to always send the delivery address (assuming Listing 7.1). This does not affect the interoperability of the client with the service since `DeliveryInfo` is optional for the PopService.

On the behavioral layer, PopClient complements the protocol of PopService by invoking it with a purchase order and awaiting for a reply. Assuming that BPEL is used

```
<types>
 <xsd:schema>
  <xsd:complexType name="PODocument">
   <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:schema>
</types>
```

Listing 7.1: PopClient Message Schema (version 1.0)

for expressing the client protocol, the BPEL process of PopClient is shown in Listing 7.2.

```
<partnerLinks>
 <partnerLink name="Service" partnerLinkType="POPServiceLinkType"
 myRole="POPClient" partnerRole="POPService"/>
  ...
</partnerLinks>

<variables>
 <variable name="PO" messageType="ns:POMessage"/>
 <variable name="POAck" messageType="ns:POMessageAck"/>
  ...
</variables>

<sequence>
   <invoke name="SubmitPOAck" partnerLink="Service"
   operation="receivePO" portType="ns:POPServicePortType"
   inputVariable="PO" createInstance="yes"/>
   ...
   <receive name="ReceivePO" partnerLink="Service"
   operation="receivePOCallBack" portType="ns:POPServiceCallBackPortType"
   variable="POAck"/>
</sequence>
</process>
```

Listing 7.2: PopClient BPEL file (version 1.0)

As for the non-functional layer, while PopClient respects the security requirements set by the service, it has more relaxed expectations about the QoS characteristics of the service, as summarized in Table 7.1. More specifically, it expects availability anywhere between 80 and 90% of the time (meaning that it can also accept larger values of availability without however obligating the service to provide them), latency between 20 (or less) and 60 seconds and minimum of 75% reliability.

Property	Value
Availability	minimum 80% and maximum 90% of the time
Latency	Minimum 20 secs, maximum 60 secs
Reliability	Minimum 75% across the board
Authentication	HMAC-SHA1 signature
Data Encryption	Base64Binary

Table 7.1: PopClient Non-functional Properties

7.2.1 ASD Representation of the Consumer

Since one of the fundamental assumptions in this work is that everything is described a service, we use the ASD notation developed in Chapter 4 to represent PopClient (that is, as we did for representing PopService). In order to distinguish the service from the client we use $\mathcal{S}_\mathcal{P}$ and p to denote the PopService ASD and its records, and $\mathcal{S}_\mathcal{C}$ and c to denote the PopClient ASD and its records, respectively. The non-functional layer of the ASD of PopClient for example consists of the records:

$$
\begin{aligned}
c_{assert_1} &= (assert_1, availability, monotonic, [80, 90], obligation) \\
c_{assert_2} &= (assert_2, latency, antitonic, [20, 60], obligation) \\
c_{assert_3} &= (assert_3, reliability, monotonic, [75, 100], obligation) \\
c_{aset_1} &= (aset_1) \\
c_{pfl_1} &= (pfl_1) \\
r(c_{aset_1}, c_{assert_1}) &= (aset_1, assert_1, AND, [1, 1]) \\
r(c_{aset_1}, c_{assert_2}) &= (aset_1, assert_2, AND, [1, 1]) \\
r(c_{aset_1}, c_{assert_3}) &= (aset_1, assert_3, AND, [1, 1]) \\
r(c_{pfl_1}, c_{aset_1}) &= (pfl_1, aset_1, OR, [1, 1])
\end{aligned}
$$

The PopClient ASD contains the same dimensions as the ASD for the PopService but with an *inversed* role: while PopService promises to offer `Availability` between 80 and 95% of the time, PopClient expects to be offered `Availability` between 80 and 90%. Records c_{assert_1}-c_{assert_3} have for this purpose the *obligation* property, as defined in Chapter 4. By these means, $\mathcal{S}_\mathcal{C}$ denotes that PopClient is expecting the other party (in that case the PopService) to respect the stated value ranges and offer the respective quality dimensions within these ranges.

A similar inversion also occurs to the other aspects of the PopClient ASD: the behavioral description of the client is similar to the behavioral description of the service but where the service is awaiting for input (in a receive `Activity`) the client is providing output (with an invoke `Activity`). The structural elements that are used an input-type message payload for PopService are output-type for PopClient (following their role in Listing

7.2) and so on. The consumer ASD can be therefore perceived as the result of a partial inversion of $\mathcal{S}_{\mathcal{P}}$ that created a "complementary" to $\mathcal{S}_{\mathcal{P}}$ ASD, $\mathcal{S}_{\mathcal{C}}$.

7.2.2 Change Scenario IV

As with the PopService, we also assume that the client evolves at some point in its life time. In particular, we assume that as part of the monitoring of the performance of the PopService conducted by the manager of the PopClient it was realized that the QoS characteristics originally required were set too low. For that reason, and in order to operate under a more realistic assumption about the operational capabilities of the service the client QoS characteristics are revised upwards as shown in Table 7.2.

Property	**Value**
Availability	minimum 80% and maximum 95% of the time
Latency	Minimum 20 secs, maximum 30 secs
Reliability	Minimum 85% across the board

Table 7.2: Change Scenario IV – PopClient Non-functional Properties

The (new) ASD for the PopClient therefore would contain the following records:

$$
\begin{aligned}
c'_{assert_1} &= (assert_1, availability, monotonic, [80, 95], obligation) \\
c'_{assert_2} &= (assert_2, latency, antitonic, [20, 30], obligation) \\
c'_{assert_3} &= (assert_3, reliability, monotonic, [85, 100], obligation) \\
c'_{aset_1} &= (aset_1) \\
c'_{pfl_1} &= (pfl_1) \\
r'(c'_{aset_1}, c'_{assert_1}) &= (aset_1, assert_1, AND, [1, 1]) \\
r'(c'_{aset_1}, c'_{assert_2}) &= (aset_1, assert_2, AND, [1, 1]) \\
r'(c'_{aset_1}, c'_{assert_3}) &= (aset_1, assert_3, AND, [1, 1]) \\
r'(c'_{pfl_1}, c'_{aset_1}) &= (pfl_1, aset_1, OR, [1, 1])
\end{aligned}
$$

It can be shown that $\Delta\mathcal{S}_{IV} \notin \mathbb{T}$, that is, the change scenario is not T-shaped according to Algorithm 1 since:

$$
\begin{aligned}
c'_{assert_1} &\leq c_{assert_1} \\
c'_{assert_2} &\leq c_{assert_2} \\
c'_{assert_3} &\leq c_{assert_3}
\end{aligned}
$$

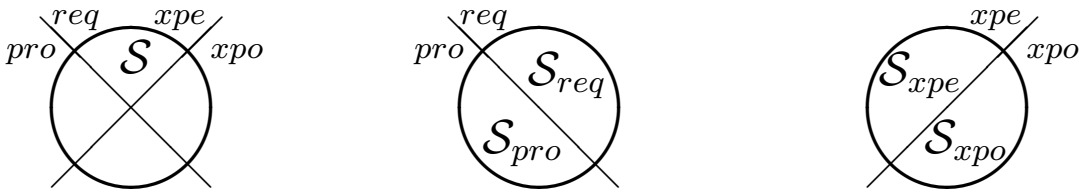
from Definition 11, and therefore $c_{pfl_1} \not\leq c'_{pfl_1}$ according to Definition 12 for `Profile` elements.

7.3 Contract Formation

The description of the PopClient exhibits a symmetry between the service provider and client with respect to the offerings and the expectations of each party. In the following we use this observation for constructing a service contract between them. Services providers and clients play different roles in an interaction (both producers and consumers of messages) and they may use more than one services for their purposes. This means that we need a way to identify which records of their ASDs are participating in the interaction, and characterize them accordingly. For that reason we define two *views* on ASDs.

7.3.1 ASD Views

We define two orthogonal *views* on the ASD $\mathcal{S}$ (Fig. 7.2): the *xpe/xpo* (*expectation/exposition*) view and the *pro/req* (*provided/required*) view:



(a) Views on $\mathcal{S}$ (b) *pro/req* View (c) *xpe/xpo* View

Figure 7.2: ASD Views

Provided/Required View This view has been already used in the previous chapter (Definition 6). In particular, the division enforced by this view (Fig. 7.2b) is quite straightforward: it provides the means to cleanly separate input from output in a service representation (irrespective of whether it acts as a provider or a client). More specifically:

- Provided $\mathcal{S}_{pro}$: contains the output-type records of the service.
- Required $\mathcal{S}_{req}$: contains the input-type records.

From Section 6.3 and for Listings 3.1, 3.2 and Table 3.1, for example, for the records of the PopService we have already established that:

$$\{p_{di}, p_{oi}, p_{pod}, r(p_{pod}, p_{di}), r(p_{pod}, p_{oi}), p_{msg}, r(p_{msg}, p_{pod}), p_{ReceivePO}\} \in \mathcal{S}_{\mathcal{P}req}$$
$$\{p_{res}, p_{poack}, r(p_{res}, p_{poack}), p_{SubmitPOAck}\} \in \mathcal{S}_{\mathcal{P}pro}$$

This distribution is *inversed* for the ASD $\mathcal{S}_{\mathcal{C}}$ of the PopClient service: since the client has to invoke the service using the `receivePO` operation with `POMessage` payload, then

the `Message` element c_{msg} is an output type for the client, that is, $c_{msg} \in \mathcal{S}_{\mathcal{C}req}$. In similar fashion, all its records in the structural and behavioral layers will be in the inverse subset with respect to $\mathcal{S}_{\mathcal{P}}$: $\{c_{di}, c_{oi}, e_{pod}, \dots\} \in \mathcal{S}_{\mathcal{C}pro}$ and $\{c_{res}, c_{poack}, r(c_{res}, c_{poack}), \dots\} \in \mathcal{S}_{\mathcal{C}req}$.

It can be deduced from the above that the *pro/req* view is *partial*. There are records of an ASD $\mathcal{S}$ that can not be classified into one of these subsets. `Protocol` elements for example, due to the fact that they may contain both input and output records are not into one of these sets. We classify these records as belonging in the $\mathcal{S}_{net}$ subset (from their *neutral* role):

- Neutral $\mathcal{S}_{net} = \mathcal{S} - \{\mathcal{S}_{pro} \cup \mathcal{S}_{req}\}$: contains the records that do not belong in the provided or required (sub)sets.

It therefore holds by definition that $\mathcal{S}_{pro} \cup \mathcal{S}_{req} \cup \mathcal{S}_{net} = \mathcal{S}$.

Expectation/Exposition view This view (Fig. 7.2c) classifies the records within an ASD with respect to whether they are offered as an interface to the environment or they are "imported" into the ASD, by referring to ASD records of other services. In the former case, the service acts as a provider; in the latter as a client of other services (both in the cases of service composition and SBA construction). Records of a service representation can therefore fall into one of the following categories:

- Exposition $\mathcal{S}^{xpo}$: the published set of records that describe the offered functionality of the service.

- Expectation $\mathcal{S}^{xpe}$: the private set of records describing the functionality offered by other service providers to the service.

The WSDL document of PopService for example in Listing 3.1 contains the information on how to access the records that constitute the Purchase Order Processing service and what information is exchanged while accessing it. From the perspective of the provider of the service, this document specifies what the provider will offer to the service customers: if the `receivePO` operation is invoked using the `POPServicePortType` and the message payload defined, the result will be an acknowledgement string. The elements of the document are in that sense in the *xpo* subset of the service provider.

On the other hand, when a consumer of this service like PopClient builds an SBA based on the service, the consumer refers to what it perceives to be a set of records that allow it to access the service. To put it simply, the client is built on the premise of a particular ASD of the provided service, being bound for example to Listing 3.1. Those records are therefore contained in the *xpe* subset of the consumer ASD. What becomes apparent from this is that the same records can either be expositions or expectations; it only depends on the adopted viewpoint.

Ideally, the *perceived* ASD and the *actual* ASD of the provided service are the same – and that is so far the fundamental assumption in service interactions. But changes to

either side, as we will discuss in the following sections, could lead to inconsistencies – in other words, incompatibilities – between those two.

In case the client is exposing functionality as a service itself, the *xpe* records are private to the extent that they are not (necessarily) published to its clients. As with the *pro*/*req* set distribution, if a record comes from the ASD of a consumed service but used as part of the *xpo* set (i.e. it is published in the ASD of the service), then the record is duplicated and appears in both sets. In contrast though to the *pro*/*req* view, this one is *complete*: $\mathcal{S}^{xpe} \cup \mathcal{S}^{xpo} = \mathcal{S}$ since a record can either be "native" to the service or "imported" to the ASD from another service.

Combining the views Since the two views are orthogonal, they can be used in conjunction to define the records of a service representation (Fig. 7.2a):

$$\mathcal{S}^{xpe} \cup \mathcal{S}^{xpo} = \mathcal{S}_{pro} \cup \mathcal{S}_{req} (\cup \mathcal{S}_{net}) = \mathcal{S}$$

In principle, only a part of the offered service functionalities may be used by a specific client; on the other hand, a client may depend on a number of disparate services in order to achieve its goals. Thus we need a way to identify and isolate the parts of the interacting parties that actually contribute to the interaction. For this purpose we will denote explicitly with $\mathcal{P} \subseteq \mathcal{S}^{xpo}_{provider}$ and $\mathcal{C} \subseteq \mathcal{S}^{xpe}_{consumer}$ the subsets of the provider and consumer ASDs respectively that participate in the interaction, as shown in Fig. 7.3.

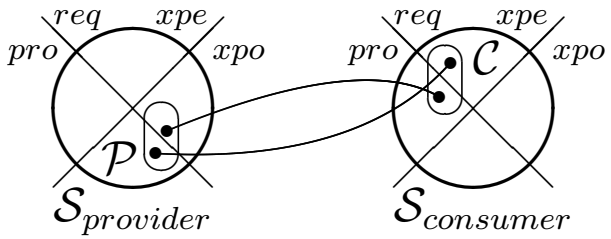


Figure 7.3: Service Interaction

In the case of the PopService and PopClient for example, we have:

- $\mathcal{S}_{\mathcal{P}}{}^{xpe} = \emptyset$ (for the PopService)
- $\mathcal{S}_{\mathcal{C}}{}^{xpo} = \emptyset$ (for the PopClient)
- $\mathcal{P} = \mathcal{S}_{\mathcal{P}}{}^{xpo}$ and $\mathcal{C} = \mathcal{S}_{\mathcal{C}}{}^{xpe}$

Fig. 7.3 exhibits a symmetry between the records $\mathcal{P}$ and $\mathcal{C}$ sets. In the following sections we are going to build on this symmetry in order to form a contract between provider $\mathcal{P}$ and consumer $\mathcal{C}$. Contract formation in our work is driven by the compatibility between the exposition records in $\mathcal{P}$ and the expectation records of $\mathcal{C}$. In Chapter 6 we developed a theory of compatibility based on the subtyping relation between records of service versions.

Since both providers and consumers are being represented in the ASD notation we can reuse this theory for formally describing this compatibility between the records of $\mathcal{P}$ and $\mathcal{C}$.

A critical observation here is that the records in the $\mathcal{C}$ set differ in principle by the respective records in the $\mathcal{P}$ set in their properties. In the case of PopService and PopClient for example, the `Assertion` elements $p_{assert_1}, p_{assert_2}, p_{assert_3}$ and $c_{assert_1}, c_{assert_2}, c_{assert_3}$ are defined on the same dimensions but from different perspectives. $p_{assert_1}, \ldots$ express the QoS characteristics offered by the PopService, while $c_{assert_1}, \ldots$ codify the expectations of PopClient with respect to the QoS of the consumed service. By the discussion on non-functional subtyping in Chapter 6 we can see that $c_{assert_1}, \ldots$ are subtypes of $p_{assert_1}, \ldots$ since the latter value ranges are more generic that the former ones. Applying the subtyping relation for assertions (Definition 11) though is not directly possible since they have different *role* properties. For this purpose we are defining the inversion operator on service records:

Definition 14

Inversion Operator

For a record $s \in \mathcal{S}$, the inversion $\overline{s}$ is defined as:

- For an element $e = (name, att_1, \ldots, att_k, pr_1, \ldots, pr_l)$ it holds:

$$\overline{e} = (name, att_1, \ldots, att_k, \overline{pr_1}, \ldots, \overline{pr_l})$$

$$\text{where } \begin{cases} \overline{input} = output \ \vee \ \overline{input} = fault \\ \overline{output} = input, \ \overline{fault} = input \\ \overline{invoke} = reply \ \vee \ \overline{invoke} = receive \\ \overline{reply} = invoke, \ \overline{receive} = invoke \\ \overline{post-} = pre-, \ \overline{pre-} = post- \\ \overline{promise} = obligation, \ \overline{obligation} = promise \\ \overline{pr_j} = pr_j \text{ otherwise} \end{cases}$$

- For a relationship $r(e_s, e_t)$ it holds:

$$\overline{r(e_s, e_t)} := r(\overline{e_s}, \overline{e_t})$$

Inverting an element $e_{msg} = (msg, input)$ for example results into $\overline{e_{msg}} = (msg, output)$ or $\overline{e_{msg}} = (msg, fault)$ (both transformations are acceptable). For `Assertion` element $e_{assert} = (assert, dimension, dimtype, [min, max], promise)$ it holds $\overline{e_{assert}} = (assert, dimension, dimtype, [min, max], obligation)$, etc. Using Definition 14 we can apply not only Definition 11, but also all the other subtyping relations we defined in the previous chapter to check for the compatibility of records in $\mathcal{P}$ with their (inversed) counterparts in $\mathcal{C}$.

The inversion operator affects also the distribution of records: if $s \in \mathcal{S}_{pro}$ then $\overline{s} \in \mathcal{S}_{req}$, and vice versa. Similarly, $s \in \mathcal{S}^{xpe} \Rightarrow \overline{s} \in \mathcal{S}^{xpo}$. Since records are characterized by a pair of $(pro/req, xpe/xpo)$ dimensions, then the operator inverses this characterization too: $s \in \mathcal{S}^{xpe}_{pro} \Rightarrow \overline{s} \in \mathcal{S}^{xpo}_{req}$, and so on. It is also possible to invert whole subsets; by writing ${\mathcal{S}_{\mathcal{P}}}^{xpo} = \overline{{\mathcal{S}_{\mathcal{C}}}^{xpe}}$ for example we denote that $\forall p \in {\mathcal{S}_{\mathcal{P}}}^{xpo}$, $\exists c \in {\mathcal{S}_{\mathcal{C}}}^{xpe} : \; p = \overline{c}$. Using the subtyping relations we developed for the evolution of services, and the inversion operator defined here we develop in the following sections a method for the matchmaking and contract configuration between service providers and consumers.

7.3.2 Matchmaking

For the purposes of matchmaking service provider and client ASDs we define a binding function ϑ that reasons *horizontally* across the records of parties $\mathcal{P}$ and $\mathcal{C}$:

Definition 15

Service Matching

A service matching is a binding function $\vartheta : \mathcal{P} \times \mathcal{C} \rightarrow \mathcal{U}, \mathcal{U} = \mathcal{P} \cup \mathcal{C}$ defined as

$$\vartheta(x,y) = \{ z \in \mathcal{U} / \begin{cases} x \leq z \leq \overline{y}, x \in \mathcal{P}_{req}, y \in \mathcal{C}_{pro} \\ \overline{y} \leq z \leq x, x \in \mathcal{P}_{pro}, y \in \mathcal{C}_{req} \\ \overline{\mathtt{y_{prt}}} \leq \mathtt{z_{prt}} \leq \mathtt{x_{prt}}, \mathtt{x_{prt}} \in \mathcal{P}, \mathtt{y_{prt}} \in \mathcal{C} \\ \overline{\mathtt{y_{pfl}}} \leq \mathtt{z_{pfl}} \leq \mathtt{x_{pfl}}, \mathtt{x_{pfl}} \in \mathcal{P}, \mathtt{y_{pfl}} \in \mathcal{C} \end{cases} \}$$

where $\mathtt{x_{prt}}, \mathtt{y_{prt}}$ are `Protocol` elements and $\mathtt{x_{pfl}}, \mathtt{y_{pfl}}$ are `Profile` elements. As with Algorithm 1 in Chapter 6, these additions are necessary since `Protocol` elements have relationships with both *pro* and *req* elements and non-functional records (`Profile`s, `Assertion`s and `Assertion Set`s) do not belong in either of the $\mathcal{S}_{pro}, \mathcal{S}_{req}$ sets.

Binding function ϑ is acting in the same manner as a *schema matching* function would. Schema matching aims at identifying semantic correspondences between elements of two schemas, e.g., database schemas, ontologies, and XML message formats [165], [166]. It is necessary in many database applications, such as integration of web data sources, data warehouse loading and XML message mapping. In most systems, schema matching is manual or semi-automatic; a time-consuming, tedious, and error-prone process which becomes increasingly impractical with a higher number of schemas and data sources to be dealt with. In our case though, the matching function relies on the subtyping relation (Definition 9 and its extensions of the behavioral and non-functional layer) to automatically identify elements on either party that are semantically related to each other according to their respective schemata.

For the structural aspect of PopService and PopClient services for example, we have:

$$\vartheta(p_{di}, c_{di}) = \{z_{di}\}, p_{di} = z_{di} = \overline{c_{di}} \qquad p_{di} \in \mathcal{P}_{req}, c_{di} \in \mathcal{C}_{pro}$$
$$\vartheta(p_{oi}, c_{oi}) = \{z_{oi}\}, p_{oi} = z_{oi} = \overline{c_{oi}} \qquad p_{oi} \in \mathcal{P}_{req}, c_{oi} \in \mathcal{C}_{pro}$$
$$\ldots$$
$$\vartheta(p_{res}, c_{res}) = \{z_{res}\}, \overline{c_{res}} = z_{res} = p_{res} \qquad c_{res} \in \mathcal{C}_{req}, p_{res} \in \mathcal{P}_{pro}$$
$$\vartheta(p_{poack}, c_{poack}) = \{z_{poack}\}, \overline{c_{poack}} = z_{poack} = p_{poack} \qquad c_{poack} \in \mathcal{C}_{req}, p_{poack} \in \mathcal{P}_{pro}$$
$$\ldots$$
$$\vartheta(r(p_{pod}, p_{di}), r(c_{pod}, c_{di})) = \{r(z_{pod}, z_{di})\},$$
$$r(p_{pod}, p_{di}) \leq r(z_{pod}, z_{di}) \leq \overline{r(c_{pod}, c_{di})} \qquad r(p_{pod}, p_{di}) \in \mathcal{P}_{req}, r(c_{pod}, c_{di}) \in \mathcal{C}_{pro}$$

where $r(z_{pod}, z_{di}) = (PODocument, DeliveryInfo, s, [n, 1]), \begin{cases} n = 0 \\ n = 1 \end{cases}$. While for the other records ϑ returns singletons (sets of one element), for the structural relationship between the purchase order document and delivery info `Information Type` it returns two possible values. Both values in the $\{r(z_{pod}, z_{di})\}$ are acceptable according to the definition of ϑ.

For the behavioral aspect we need to check the elements $p_{ReceivePO} \in \mathcal{S}_{\mathcal{P}req}, p_{SubmitPOAck} \in \mathcal{S}_{\mathcal{P}pro}$ and their inversions in the client $c_{SubmitPOAck} \in \mathcal{S}_{\mathcal{C}pro}, c_{ReceivePO} \in \mathcal{S}_{\mathcal{C}req}$, and the protocols p_{seq} and c_{seq}:

$$\vartheta(p_{ReceivePO}, c_{ReceivePO}) = \{z_{ReceivePO}\}, p_{ReceivePO} = z_{ReceivePO} = \overline{c_{ReceivePO}}$$
$$\vartheta(p_{SubmitPOAck}, c_{SubmitPOAck}) = \{z_{SubmitPOAck}\}, \overline{c_{SubmitPOAck}} = z_{SubmitPOAck} = p_{SubmitPOAck}$$
$$\vartheta(p_{seq}, c_{seq}) = \{z_{seq}\}, \overline{c_{seq}} = z_{seq} = p_{seq} \text{ since } \sigma(\overline{c_{seq}}) = \sigma(p_{seq})$$

So far, and due to the fact that PopClient is using PopService "as-is", the subtyping relation in Definition 15 resulted almost always in equalities. Due to the difference between the QoS characteristics expected from PopClient and the ones offered by PopService, applying the ϑ to the `Assertion` elements results in sets with more than one value. In order to demonstrate this, we start by looking at the relations of the `Assertion` elements in sets $\mathcal{P}$ and $\mathcal{C}$ using the definition of assertion subtyping (Definition 11) from Chapter 6:

$$assert_1 = assert_1 \wedge availability = availability \wedge monotonic = monotonic$$
$$\wedge promise = \overline{obligation} \wedge [80, 90] \ \mathsf{s} \ [80, 95] \Rightarrow \overline{c_{assert_1}} \leq p_{assert_1}$$
$$assert_2 = assert_2 \wedge latency = latency \wedge antitonic = antitonic$$
$$\wedge promise = \overline{obligation} \wedge [20, 60] \ \mathsf{oi} \ [15, 30] \Rightarrow \overline{c_{assert_2}} \leq p_{assert_2}$$
$$assert_3 = assert_3 \wedge reliability = reliability \wedge monotonic = monotonic$$
$$\wedge promise = \overline{obligation} \wedge [75, 100] \ \mathsf{fi} \ [90, 100] \Rightarrow \overline{c_{assert_3}} \leq p_{assert_3}$$

From these, and from Definition 12 for the subtyping of `Assertion Set` and `Profile` elements, we can conclude that $\overline{c_{pfl_1}} \leq p_{pfl_1}$ since it holds $\overline{c_{assert_1}} \leq p_{assert_1} \wedge \overline{c_{assert_2}} \leq p_{assert_2} \wedge \overline{c_{assert_3}} \leq p_{assert_3} \Rightarrow \overline{c_{aset_1}} \leq p_{aset_1} \Rightarrow \overline{c_{pfl_1}} \leq p_{pfl_1}$. Definition 11 actually provides many choices in picking the value of z in Definition 15. As long as we can satisfy the $\overline{y} \leq z \leq x$ conditions we can allow any value range for the `Assertion` represented by z. In specific:

$$\begin{aligned}
\vartheta(p_{assert_1}, c_{assert_1}) &= \{z_{assert_1}\}, \overline{c_{assert_1}} \leq z_{assert_1} \leq p_{assert_1} \qquad p_{assert_1} \in \mathcal{P}, c_{assert_1} \in \mathcal{C} \\
\vartheta(p_{assert_2}, c_{assert_2}) &= \{z_{assert_2}\}, \overline{c_{assert_2}} \leq z_{assert_2} \leq p_{assert_2} \qquad p_{assert_2} \in \mathcal{P}, c_{assert_2} \in \mathcal{C} \\
\vartheta(p_{assert_3}, c_{assert_3}) &= \{z_{assert_3}\}, \overline{c_{assert_3}} \leq z_{assert_3} \leq p_{assert_3} \qquad p_{assert_3} \in \mathcal{P}, c_{assert_3} \in \mathcal{C}
\end{aligned}$$

where

$$\begin{aligned}
z_{assert_1} &= (assert_1, availability, monotonic, [min, max], promise), \\
&\quad min \in [80, 90],\ max \in [90, 95] \\
z_{assert_2} &= (assert_2, latency, antitonic, [min, max], promise), \\
&\quad min \in [15, 20],\ max \in [30, 60] \\
z_{assert_3} &= (assert_3, reliability, monotonic, [min, max], promise), \\
&\quad min \in [75, 90],\ max \in [100, 100]
\end{aligned}$$

For $z_{assert_2} = (assert_2, latency, antitonic, [15, 40], promise)$ for example it holds that

$$\left.\begin{array}{r}
assert_2 = assert_2 \wedge latency = latency \wedge antitonic = antitonic \\
\wedge promise = \overline{obligation} \wedge [15, 40]\ \mathsf{oi}\ [20, 60] \Rightarrow \overline{c_{assert_2}} \leq z_{assert_2} \\
assert_2 = assert_2 \wedge latency = latency \wedge antitonic = antitonic \\
\wedge promise = promise \wedge [15, 40]\ \mathsf{si}\ [15, 30] \Rightarrow z_{assert_2} \leq p_{assert_2}
\end{array}\right\} \Rightarrow \overline{c_{assert_2}} \leq z_{assert_2} \leq p_{assert_2}$$

This flexibility in choosing the actual values for the binding function ϑ allows us to define different contract configuration policies as we discuss in the following.

7.3.3 Contract Configuration

Based on the binding function ϑ we can define the *Contract* $\mathcal{R}$ between two parties as a service mapping:

Definition 16

Service Mapping

A service mapping is a *Contract* $\mathcal{R}$ defined by a triplet $\mathcal{R} =< \mathcal{P}, \mathcal{C}, \Theta >$ between two parties $\mathcal{P}$ and $\mathcal{C}$, where Θ is defined as the image of $\mathcal{P}$ and $\mathcal{C}$ under ϑ, i.e. $\Theta = \{\vartheta(p, c)/p \in \mathcal{P}, c \in \mathcal{C}\}$. The records z that comprise $\mathcal{R}$ are called the *clauses* of the contract.

The service mapping therefore consists of the results of the service matching for all possible record pairs in the provider/client ASDs and is formulated by reasoning vertically through the parties. The contract that is produced by this mapping identifies and represents the mutually agreed ASD records that will be used for the interaction of the parties. Figure 7.4 demonstrates the relation between $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{R}$ graphically.

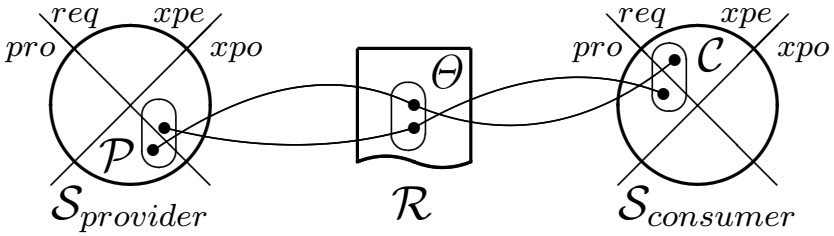


Figure 7.4: Contract Configuration

The definition of contract $\mathcal{R}$ between two parties as a service mapping $< \mathcal{P}, \mathcal{C}, \Theta >$ allows for a straightforward formulation of the contract: given the two parties' ASDs $\mathcal{P}$ and $\mathcal{C}$, each of which defines the records through which the interaction is achieved, Θ can be calculated directly by applying the binding function ϑ to them. Contract formation therefore implicitly depends on producing $\mathcal{P}$ and $\mathcal{C}$ from the service provider $\mathcal{S}^{xpo}_{provider}$ and client $\mathcal{S}^{xpe}_{consumer}$ ASDs respectively.

Due to the fact that the service provider is unaware of the internal workings of the service client (represented by the $\mathcal{S}^{xpe}_{consumer}$ set) the process of contract formation is consumer-driven; more specifically, the steps to be followed are shown in the *Contract Formation Algorithm* (Algorithm 2).

Table 7.3 shows one of the possible contracts that can be formed between PopService and PopClient. The direction of the subtyping relation here depends on the distribution of the records of the services in the *pro*/*req* subsets. The formation, storing and reasoning aspects of the proposed solution can be incorporated in the service governance infrastructure that supports each party. In that respect, contract formation is an aspect of service governance.

7.3.4 Configuration Policies

Since ϑ may return one or more possible values, depending on the subtyping "distance" of the records in $\mathcal{P}$ and $\mathcal{C}$, a minimum level of insight on the client side is required in selecting values from the binding function ϑ for the construction of Θ. Different policies of the configuration of the contract are possible:

Conservative selection policies opt for the values contributed by the client to the calculation of ϑ, trying to protect the client from possible changes to the producer.

Liberal selection policies on the other hand pick the values contributed by the provider and allow for the possibility of the client evolving more freely in the future.

Algorithm 2 Contract Formation Algorithm

1. The client decides on the functionality offered by the provider that will be used (if more than one is offered).
2. The set of records from $\mathcal{S}^{xpo}_{provider}$ that fulfill this functionality are identified and associated with the $\mathcal{P}$ set.
3. The identified records are either copied to the (initially empty) $\mathcal{C} = \mathcal{S}^{xpe}_{consumer}$ set or the existing $\mathcal{C}$ set is used.
4. The image of $\mathcal{P}$ and $\mathcal{C}$ under ϑ is calculated. If the resulting set is empty then the image is attempted to be re-calculated using alternative values from ϑ (or canceled, in case all possibilities have been exhausted); otherwise the contract $\mathcal{R} =< \mathcal{P}, \mathcal{C}, \Theta >$ is produced.
5. The consumer submits the formulated contract $\mathcal{R}$ to the producer for posterity and begins interaction with provider.

Mixed selection policies combine values from the provider's and client's side.

For the contract between PopService and PopClient in Table 7.3 for example, we opted for a mixed policy, allowing values from both the provider and the client to appear in the contract $\mathcal{R}$. The type of policy to be followed is therefore largely a design and governance issue and has to be dealt as such. The solution presented assumes that producers and consumers have the means to form, exchange, store, and reason on the basis of contracts. In absence of these facilities from one or both parties the interaction between them reverts to the non contract-based modus operandi, using only T-shaped changes. The exchange of contracts requires the existence of a dedicated mechanism for this purpose that is not part of the service representation.

7.4 Service Evolution with Contracts

The previous sections discussed how to form a contract between interacting parties in an atemporal manner – similarly to the representation of a service by a non-versioned ASD. In the following we introduce the evolution of the parties in the equation. We show how service contracts are allowing for more flexibility in the evolution of services, and how they can themselves evolve while facilitating the interoperability between providers and clients.

In particular, in the initial 'static' state of two interoperating parties $\mathcal{P}$ and $\mathcal{C}$, and after a contract $\mathcal{R} =< \mathcal{P}, \mathcal{C}, \Theta >$ has been formed and accepted between them, it holds in general that $\mathcal{P} \equiv \overline{\mathcal{C}}$ (assuming a very simple client), and by the definition of the contract construct, $\mathcal{P} \equiv \Theta \equiv \overline{\mathcal{C}}$, as we have seen in the previous section. But since either party can, or at least

Layer	$\mathcal{P}$		$\mathcal{R}$		$\overline{\mathcal{C}}$
Structural	p_{di}	$\leq$	p_{di}	$\leq$	$\overline{c_{di}}$
	p_{pod}	$\leq$	p_{pod}	$\leq$	$\overline{c_{pod}}$
	$r(p_{pod}, p_{di})$	$\leq$	$r(p_{pod}, p_{di})$	$\leq$	$\overline{r(c_{pod}, c_{di})}$
			...		
Behavioral	$p_{ReceivePO}$	$\leq$	$p_{ReceivePO}$	$\leq$	$\overline{c_{ReceivePO}}$
	$p_{SubmitPOAck}$	$\geq$	$p_{SubmitPOAck}$	$\geq$	$\overline{c_{SubmitPOAck}}$
	p_{seq}	$\geq$	p_{seq}	$\geq$	$\overline{c_{seq}}$
Non-functional	p_{assert_1}	$\geq$	$\overline{c_{assert_1}}$	$\geq$	$\overline{c_{assert_1}}$
	p_{assert_2}	$\geq$	$\overline{c_{assert_2}}$	$\geq$	$\overline{c_{assert_2}}$
	p_{assert_3}	$\geq$	$\overline{c_{assert_3}}$	$\geq$	$\overline{c_{assert_3}}$

Table 7.3: Contract example between PopService & PopClient

should be able to, evolve independently of the other, shifts from this state can occur. When changes for example occur to the provider, then it may hold that $\mathcal{P}' \not\equiv \Theta \equiv \overline{\mathcal{C}}$, or for the consumer side $\mathcal{P} \equiv \Theta \not\equiv \overline{\mathcal{C}'}$, or both. These latter states reflect situations of incompatibility between producer and consumer and they have to be prevented from occurring in order to avoid the occurrence of deep changes in the context of the interacting parties.

The introduction of a contract between them allows us to reason about the contribution of each party to the interaction without directly affecting the other party, ensuring that each party is able to evolve independently but transparently, that is without requiring modifications, to each other. In this sense, version of the parties that comply to the contract between them are shallow, irrespective of whether they are compatible to the previous version or not according to Definition 7 in Chapter 6.

7.4.1 Contractually-bound Evolution

Taking advantage of the ability to reason exclusively on one party given an existing contract, without the need for the other party to participate in this reasoning, exemplifies the notion of independence in evolution. In order to show how this is accomplished we will first formally define what it means for an evolving party to respect, or to be *compliant* with an existing contract:

Definition 17

Compliance to Contract

A party, e.g. provider $\mathcal{P}'$, is said to be *compliant* to a contract $\mathcal{R} =< \mathcal{P}, \mathcal{C}, \Theta >$ with a consumer $\mathcal{C}$ denoted by $\mathcal{P}' \vDash_{\mathcal{R}} \mathcal{C}$ iff

$$\forall z \in \Theta / \exists p' \in \mathcal{P}', \vartheta(p', c) = z, c \in \mathcal{C}$$

Corollary: $\mathcal{P}'$ *violates* $\mathcal{R}$, and we write $\mathcal{P}' \nvDash_{\mathcal{R}} \mathcal{C}$, iff $\exists z \in \Theta / \forall p' \in \mathcal{P}', \vartheta(p', c) \neq z, c \in \mathcal{C}$.

This definition allows for a simple algorithm to check for the compliance of a new version of a party in the producer-consumer relationship: as long as there is a mapping produced by ϑ to *all* clauses of the contract from the elements of the new specification, the two versions are equivalent or *compatible* with respect to the contract – or more formally:

Definition 18

Contract-based Compatibility

A service contract $\mathcal{R}$ is called

1. *backward compatible* and we write $\mathcal{C} \mapsto_{\mathcal{R}} \mathcal{C}'$ iff $\mathcal{P} \vDash_{\mathcal{R}} \mathcal{C} \wedge \mathcal{P} \vDash_{\mathcal{R}} \mathcal{C}'$,
2. *forward compatible* and we write $\mathcal{P} \mapsto_{\mathcal{R}} \mathcal{P}'$ iff $\mathcal{P} \vDash_{\mathcal{R}} \mathcal{C} \wedge \mathcal{P}' \vDash_{\mathcal{R}} \mathcal{C}$, and
3. *(fully) compatible* iff it is both backward and forward compatible:

$$\mathcal{C} \mapsto_{\mathcal{R}} \mathcal{C}' \wedge \mathcal{P} \mapsto_{\mathcal{R}} \mathcal{P}'$$

Lemma T-shaped change sets always lead to compatible service contracts:

$$\forall \Delta\mathcal{P} \in \mathbb{T} \Rightarrow \mathcal{P} \mapsto_{\mathcal{R}} \mathcal{P} \circ \Delta\mathcal{P}$$

and

$$\forall \Delta\mathcal{C} \in \mathbb{T} \Rightarrow \mathcal{C} \mapsto_{\mathcal{R}} \mathcal{C} \circ \Delta\mathcal{C}$$

Proof. The truth of this lemma can be shown constructively by starting from $\mathcal{P}$ and assuming a T-shaped change set $\Delta\mathcal{P}$. From Algorithm 1 we know that $\forall p' \in \mathcal{P}'_{req}, \exists p \in \mathcal{P}_{req}, p \leq p'$ and consequently, by Definition 15, $z \in \vartheta(p', c), z = \vartheta(p, c)$ since $c \leq z \leq p \leq p'$. Therefore, $\forall p' \in \mathcal{P}'_{pro}$ it holds $\forall z \in \Theta / \exists p' \in \mathcal{P}', \vartheta(p', c) = z, c \in \mathcal{C}$ – and working in a similar manner if p is a `Protocol` or `Profile` element. By its definition then, $\mathcal{P}' \vDash_{\mathcal{R}} \mathcal{C}$. Similarly, for backward compatibility-preserving changes it holds $\Delta\mathcal{C} \in \mathbb{T} \Rightarrow \mathcal{C}' \vDash_{\mathcal{R}} \mathcal{P}$. □

Layer	$\mathcal{P}'$		$\mathcal{R}$		$\overline{\mathcal{C}}$
Structural	$r'(p_{pod}, p_{di}) \leq r(p_{pod}, p_{di})$	$\leq$	$r(p_{pod}, p_{di})$	$\leq$	$\overline{r(c_{pod}, c_{di})}$
			...		
Non-functional	p_{assert_1}	$\geq$	$\overline{c_{assert_1}}$	$\geq$	$\overline{c_{assert_1}}$
	$p'_{assert_2} \geq p_{assert_2}$	$\geq$	$\overline{c_{assert_2}}$	$\geq$	$\overline{c_{assert_2}}$
	$p_{assert_3} \geq p'_{assert_3}$	$\geq$	$\overline{c_{assert_3}}$	$\geq$	$\overline{c_{assert_3}}$

Table 7.4: Change Scenario I – using the Contract of Table 7.3

Compliance to service contract is therefore a more general notion that service compatibility. Table 7.4 for example demonstrates the effect of applying Change Scenario I as

defined in Chapter 3 to PopService. As we discussed in the previous chapter, $\Delta\mathcal{S}_I \notin \mathbb{T}$ because $r'(p_{pod}, p_{di}) \leq r(p_{pod}, p_{di})$ and $p'_{assert_3} \leq p_{assert_3}$ (as also shown in Table 7.4). Nevertheless, it can be seen that the contract $\mathcal{R}$ formed in Table 7.3 is forward compatible with respect to the changed service ASD $\mathcal{P}'$ since

1. $r'(p_{pod}, p_{di}) \leq \overline{r(c_{pod}, c_{di})} \Rightarrow \exists z / r'(p_{pod}, p_{di}) \leq z \leq \overline{r(c_{pod}, c_{di})}$, and
2. $\overline{c_{assert_1}} \leq p_{assert_1} \wedge \overline{c_{assert_2}} \leq p'_{assert_2} \wedge \overline{c_{assert_3}} \leq p'_{assert_3} \Rightarrow \overline{c_{aset_1}} \leq p'_{aset_1} \Rightarrow \overline{c_{pfl_1}} \leq p'_{pfl_1}$ and therefore $\exists z / \overline{c_{pfl_1}} \leq z \leq p'_{pfl_1}$.

This means that for at least the particular client (PopClient) $\Delta\mathcal{S}_I$ can be applied to PopService without any side-effect. $\Delta\mathcal{S}_I$ is therefore shallow under the condition that all existing clients are compliant with service contract $\mathcal{R}$.

7.4.2 Contract Evolution

The previous section discussed the criteria under which changes to one party can leave the contract between them intact, essentially ensuring that these changes are shallow. This does not necessarily mean that all changes that do not respect this criteria are deep. The existing interaction between the parties can be preserved in certain cases despite the necessity to modify the contract due to changes to one or both of the parties involved. Contracts can therefore be compatible with each other too:

Definition 19

Contract Compatibility

A contract $\mathcal{R}'$ is called

1. *backward compatible with respect to (w.r.t.)* another contract $\mathcal{R}$ and we write $\mathcal{R} \mapsto_b \mathcal{R}'$ iff $\forall p \in \mathcal{P} / \exists z' \in \Theta', \exists c' \in \mathcal{C}', z' = \vartheta(p, c')$,
2. *forward compatible w.r.t.* another contract $\mathcal{R}$ and we write $\mathcal{R} \mapsto_f \mathcal{R}'$ iff $\forall c \in \mathcal{C} / \exists z' \in \Theta', \exists p' \in \mathcal{P}', z' = \vartheta(p', y)$, and
3. *(fully) compatible w.r.t.* another contract $\mathcal{R}$ and we write $\mathcal{R} \mapsto_c \mathcal{R}'$ iff it is both backward and forward compatible: $\mathcal{R} \mapsto_c \mathcal{R}' \Leftrightarrow \mathcal{R} \mapsto_b \mathcal{R}' \wedge \mathcal{R} \mapsto_f \mathcal{R}'$

Contracts are therefore backward/forward or simply compatible with respect to another contract if they contain compatible (in the sense of subtyping) terms. Fig. 7.5 illustrates the case of backward compatibility.

Applying for example Change Scenario IV to PopClient leads to a non-compatible contract $\mathcal{R}$ as shown in Table 7.5. The stricter QoS characteristics imposed by the client are breaking the compliance with the contract and for that reason they should not be allowed according to the discussion in the previous section. It becomes though possible to allow this change if a new version of the contract itself is formed and exchanged between parties.

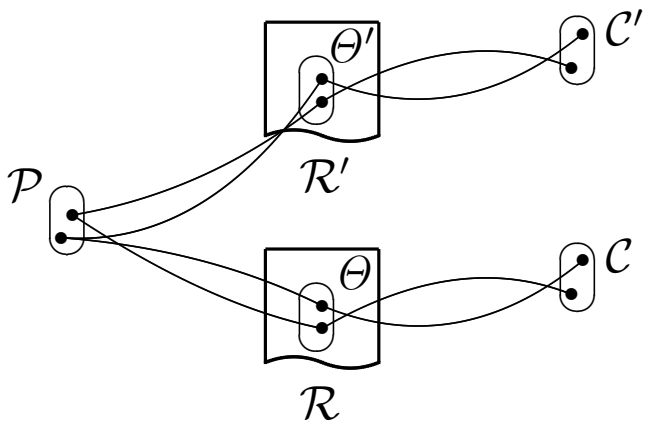


Figure 7.5: Contract Evolution – Backward Compatibility

Layer	$\mathcal{P}$		$\mathcal{R}$		$\overline{\mathcal{C}}'$
Non-functional	p_{assert_1}	$\geq$	$\overline{c_{assert_1}}$	$\leq$	$\overline{c_{assert_1}}'$
	p_{assert_2}	$\geq$	$\overline{c_{assert_2}}$	$\leq$	$\overline{c_{assert_2}}'$
	p_{assert_3}	$\geq$	$\overline{c_{assert_3}}$	$\leq$	$\overline{c_{assert_3}}'$

Table 7.5: Change Scenario IV – Contract breaking

As shown in Table 7.6, it is possible to re-generate the contract between PopService and PopClient in order to allow for the modification of the client. It holds that $\mathcal{R} \mapsto_b \mathcal{R}'$ and therefore $\mathcal{P}$ can interoperate with $\mathcal{C}'$ without any changes on any side (apart from modifying the contract between them), as shown in Fig. 7.5. Allowing therefore to evolve the contract between them allows for even greater degrees of flexibility in the evolution of the services – always at the expense of overhead though.

Layer	$\mathcal{P}$		$\mathcal{R}'$		$\overline{\mathcal{C}}'$
Non-functional	p_{assert_1}	$\geq$	$\overline{c_{assert_1}}'$	$\geq$	$\overline{c_{assert_1}}'$
	p_{assert_2}	$\geq$	$\overline{c_{assert_2}}'$	$\geq$	$\overline{c_{assert_2}}'$
	p_{assert_3}	$\geq$	$\overline{c_{assert_3}}'$	$\geq$	$\overline{c_{assert_3}}'$

Table 7.6: Change Scenario IV – Evolution of the Contract

Contrary to the case of contractually-bound evolution of the interacting parties, evolution of the contract itself requires of the parties to exchange a new contract and replace the old contract with the new one. This creates an additional communication overhead that nevertheless has to be weighted against the cost of possible inconsistencies in the current and future interactions of the parties due to discrepancies between contract versions.

7.5 Discussion

The introduction of contracts between interacting service providers and clients discussed in the previous sections is essentially an alternative model of service consumption. Instead of the one (service)-to-many (consumers) model adopted by the major services description languages and consequently by the technology vendors, service contracts promote a many-to-many model of interaction. Each consumer – or each cluster of consumers, if it it is assumed that their contracts can be aggregated based on overlap – has a separate contract with the service provider, in addition to the published service description.

Maintaining information about the expectations of each consumer allows as described for additional flexibility in the evolution of both the provider and the consumers. Depending on these expectations, more cases can be classified as shallow than those that we could deduce using (only) T-shaped changes. As a matter of fact, there is as much leeway for evolving the service provider as the difference between the expectations of the client and the expositions of the provider. Even the contract itself can evolve to accommodate the needs of either party under certain conditions.

Furthermore, the method of contract formation that we presented can be fully automated with the addition of a mechanism for choosing values for the binding function ϑ. Assuming that negotiation has already occurred in order to define the QoS characteristics to be used [167], and the complete requirements of the consumer are expressed in ASD notation, Algorithm 2 returns one (or even more) possible contracts between the parties. These contracts can be further used for monitoring and compliance enforcement if SLAs are not in place (or in addition to them).

On the downside though, this model of interaction comes with its own disadvantages. The formation of a contract with each consumer essentially increases (instead of decreasing) the coupling of the service with them, as pointed out by [9]. Contract formation implicitly increases the amount of information assumed by each party in the interaction and exposes a part of the inner workings of the consumer to the provider. Allowing as much flexibility as the consumer can handle means also that the service provider is bound to the consumers' needs instead of the service owner's. This results in loss of autonomy on the provider's side. Preserving this feature may be deemed more important than flexibility by the service provider.

Furthermore, it can be easily seen that the required reasoning on a per client basis does not scale with the number of clients. Even by grouping the consumers into clusters, the amount of contracts to be created and maintained tends to grow dramatically with the amount of evolving consumers. A robust service governance infrastructure has to already be in place in order to facilitate the management of all these contracts. This infrastructural overhead has to be added to the one for forming and exchanging contracts on both the provider and consumer side. The cost of such an overhead is not negligible and may be a serious obstacle in applying the contract-based interaction of services as discussed here.

On a closing note, a potentially interesting extension to the service contracts discussed above is the introduction of temporal conditions of availability to them. As in [114], service providers can include in the contract information about the expected life time and

the deprecation policy of the service version that they are using. This information can be critical in the case of service compositions where the decommissioning of a consumed service may require the reengineering of the composition from scratch.

7.6 Summary

This chapter opened with a brief introduction to another overloaded with definitions term, that of service contracts. While different takes on what a service contract entails were presented, the definition adopted was that of a previous work of ours. A contract in this context is an intermediary between service providers and consumers, manifesting in the form of an ASD representation. We discussed the life cycle of such a contract using the life cycle of SLAs as a guide and we scoped the discussion to the contract formation stage.

In particular, we showed how we can reuse the service representation and ASD subtyping we defined in the previous chapters in order to perform the matchmaking, provider selection (implicitly as part of the matchmaking) and contract configuration stages of the contract formation. Using the example of a consumer PopClient for the PopService, we showed how the ASD can be fragmented under different views depending on the purpose of each record and how subtyping can be used to automatically match the records of the provider's and consumer's ASDs.

This matching was then integrated into a service mapping process that configures a contract $\mathcal{R}$ between the provider and the client. Different configuration policies and their impact in this process were discussed as part of contract formation. Having established the framework for producing contracts we showed how these contracts can leverage the evolution of both providers and consumers by allowing more flexibility than previously assumed under the T-shaped property. It was also shown that under certain conditions, even the contract itself can evolve without affecting the interacting parties.

Finally, we performed an evaluation of the proposal and we concluded that while the benefits of the introduction of contracts are many, the trade-off required is not acceptable in the general case. The model for the interaction and evolution of services discussed in this chapter can be applied to organizations with a robust service governance support mechanism in place. In that sense it can not replace the compatibility theory we presented in the previous chapter but only supplement it.

Chapter 8

Validation

> The ultimate, the most sacred form of theory is action.
>
> *Nikos Kazantzakis*

> What are the figures, what are the facts, what do people mean when they talk about things?
>
> *as heard on the Monty Python's Flying Circus*

In this chapter we aim to validate the compatible service evolution model that we proposed in Chapter 6. As we discussed in the introduction (Chapter 1), the validation of our work is performed on three levels:

1. The logical consistency level, ensured by the formal underpinnings of the proposal.
2. The usability of the approach, which is exhibited by the use of a running scenario throughout all the previous chapters.
3. The realization of the solution.

This last type of validation is performed by means of proof-of-concept prototyping and by experimentation while replicating the theoretical results. Since the other validation types have been performed as part of the discussion in the previous chapters, in this one we focus on the realization aspect.

More specifically, we start the discussion by presenting our prototype implementation of a tool for the modeling and compatibility checking of service ASD versions. We then use this implementation as part of a validation experiment that enables us to confirm the feasibility of our proposal and its applicability to current Web services standards. Towards that goal we reuse the change scenarios presented in Chapter 2 that we referred to throughout the previous chapters. Since we need to compare our findings with the backward compatibility guidelines for reference (Table 6.1 in Chapter 6), the emphasis is on the structural layer.

8.1 Prototype

For the implementation of the Service Representation Modeler (SRM) tool we decided to use widely supported, open source and free tools. The SRM provides two key facilities required for the experimental validation of our proposal [168]: a graphic editor for defining ASD models of service versions and a reasoning module that compares two ASD models and checks them for compatibility as discussed in Chapter 6.

We start the presentation of the SRM tool by discussing the underlying technologies that power the prototype, before showing how these technologies are put to use for its implementation and presenting its functionality.

8.1.1 Underlying Technologies

Eclipse[1] is a free, open-source, cross-platform, multi-language software development platform providing an integrated development environment and an extensible plug-in system. It is written mainly in Java but it allows the development of applications in many different languages by means of the various plug-ins. One of those plug-ins is the Eclipse Modeling Framework (EMF) [169]. EMF is a modeling framework and code generation facility that enables application development based on structured data models. EMF provides the tools and runtime support to produce Java classes for the model, along with a set of adapter classes that enable the viewing and editing of the model.

EMF models are defined in the *ecore* format, which is essentially a wrapper for an XML Metadata Interchange (XMI) document. XMI[2] is an Object Management Group (OMG) standard for exchanging meta-data information in XML. The most common use of XMI is for serializing UML models, but it can also be used (as in our case) for the serialization of models of other languages (meta-models) too. One of the ways of defining EMF models is the Emfatic language that provides a simple textual syntax for this purpose. Based on an ecore meta-model, EMF allows in conjunction with the Graphical Modeling Framework (GMF) plug-in to automatically generate editors for handling models of the language described by the ecore file.

Given the low-level operations provided by the EMF, it was opted to overlay it with the Epsilon[3] plug-in that provides model-specific tasks such as model validation, comparison and model-to-model transformation. Epsilon allows for the injection of EMF ecore models in textual form (using the Emfatic specification language) into pure ecore meta-models. It also incorporates the EuGENia tool that automatically generates the models required for implementing a GMF editor from an annotated ecore meta-model.

This set of technologies formed the basis for the prototype implementation of the SRM tool, discussed in the following.

[1] `http://www.eclipse.org/`
[2] `http://www.omg.org/technology/documents/formal/xmi.htm`
[3] `http://www.eclipse.org/gmt/epsilon/`

8.1.2 Implementation

The first step for the implementation of the SRM tool consists of the definition of the meta-model to be used for the representation of the services in the prototype. For that purpose we used the bottom layer of the ASD Meta-model (Fig. 4.1). The various elements and relationships of the structural layer were encoded in the Emfatic language as shown in Listing 8.1. We annotated the Emfatic specification of the meta-model with GMF-specific instructions that are used in the latter stages for generating the graphical editor aspect of the SRM.

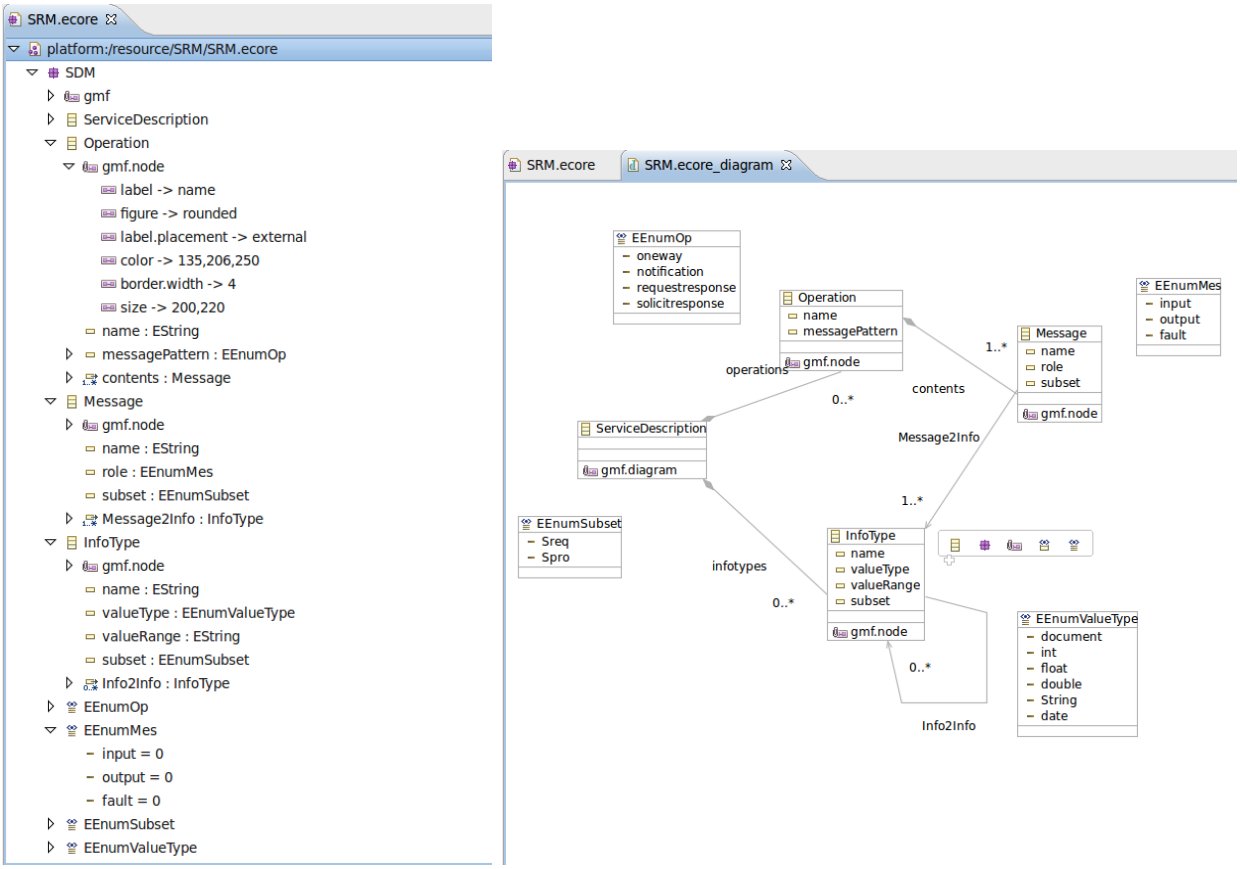
```
@gmf.node(label="name", figure="rounded", label.placement="external", color=
    "135,206,250", border.width="4", size="200,220")
class Operation {
  attr String name;  attr EEnumOp messagePattern;
  @gmf.compartment(foo="bar", layout="list")
  val Message[+] contents;
}

@gmf.node(label="name", figure="rectangle", color="193,255,193")
class Message {
  attr String name;  attr EEnumMes role;
  @gmf.link(target.decoration="arrow", style="dot", tool.description="
      Relationship␣between␣Message␣and␣InfoType")
  ref InfoType[+] Message2Info;
}
```

Listing 8.1: Emfatic specification of the ASD Meta-model (fragment)

We then used the injection facilities of Epsilon to convert the Emfatic specification of the meta-model into an ecore-type meta-model that can in turn be translated into a number of different views using the EuGENia tool. Fig. 8.1a shows for example the EMF tree editor view of the meta-model, while Fig. 8.1b shows the UML class diagram representation of the meta-model (that confirms the validity of the encoded meta-model when compared with Fig. 4.1). Based on this ecore meta-model, we automatically generated the Java code required for supporting the graphical editor of ASD models and the reasoning module. These two facilities of the SRM tool are presented in the following.

8.1.3 Functionality

Fig. 8.2 shows an example of the ASD model of a service (more specifically, that of the PopService) loaded in the graphical editor of the SRM prototype. The editor was automatically generated by EuGENia and provides the tools for creating and modifying a graphical representation of ASDs. It achieves this by offering a *Palette* panel (on the right-hand side of Fig. 8.2) containing widgets corresponding to the various structural elements in the ASD Meta-model. By selecting one of these widgets and pointing in the white canvas area the respective element is added to the ASD model of the service. Adding the name,



(a) EMF editor view (b) UML class diagram view

Figure 8.1: SRM Meta-model in ecore format

properties, relationships and the $\mathcal{S}_{pro}/\mathcal{S}_{req}$ distribution of the element is done through the *Properties* perspective (at the bottom of Fig. 8.2).

The consistency of each ASD model (as discussed in Chapter 4) can be validated against the ASD Meta-model by the use of the action menu, accessible for example in Linux by right-clicking the respective model_diagram component in the navigation panel (left part of Fig. 8.2) and opting for the *Validation* action. Unfortunately the graphical editor in its current state does not allow for the automatic transformation of WSDL documents into ASD models. All service models discussed in the following were created manually through the editor. We are currently though working on this functionality.

The reasoning module of the SRM tool was implemented as a fully functional Epsilon program. We started by translating Algorithm 1 into a set of rules for the records of the

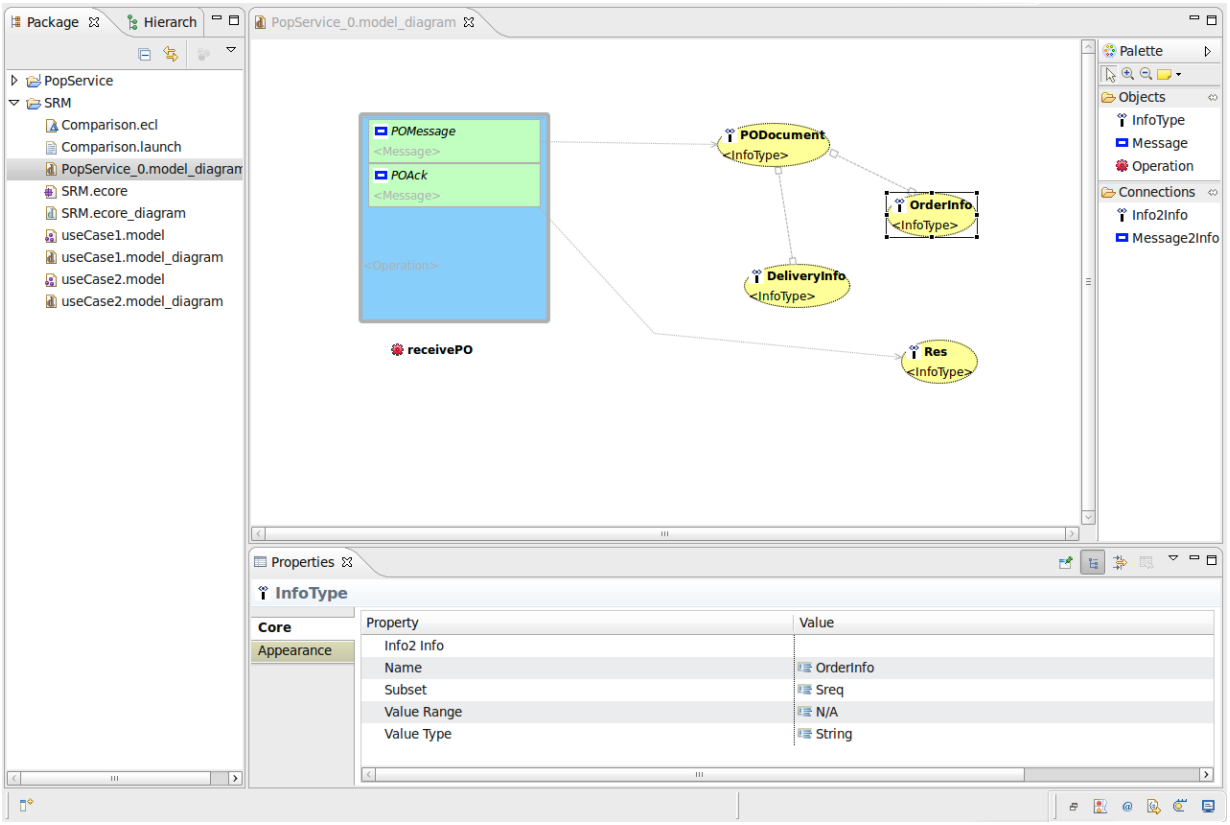


Figure 8.2: SRM prototype – graphical editor

structural layer as follows:

$$op \leq op' \Leftrightarrow name = name' \wedge messagePattern = messagePattern'$$
$$msg \leq msg' \Leftrightarrow name = name' \wedge role = role'$$
$$r(op, msg) \leq r'(op', msg') \Leftrightarrow op \leq op' \wedge msg \leq msg' \wedge mul \subseteq mul'$$
$$\ldots$$

This unrolling of the rules allowed us to encode the compatibility checking in an algorithmical manner and provide it as a module of the SRM prototype. The module takes as input two ASD models and compares them, checking for compatibility as shown in Fig. 8.3. The results are currently returned in the Epsilon console perspective inside Eclipse but we are currently working on exporting them in XML format and visualizing them using the graphical editor.

Fig. 8.3 shows the results of such a comparison between two ASDs, checking for compatibility on a record-per-record basis. If all checks are successful then the reasoner concludes with a *Pass*, otherwise it returns *Fail*. In the particular case the reasoning module returned a Fail since an `Information Type` to `Information Type` relationship was found to violate Algorithm 1. In the following section we are using this facility to confirm that the theoret-

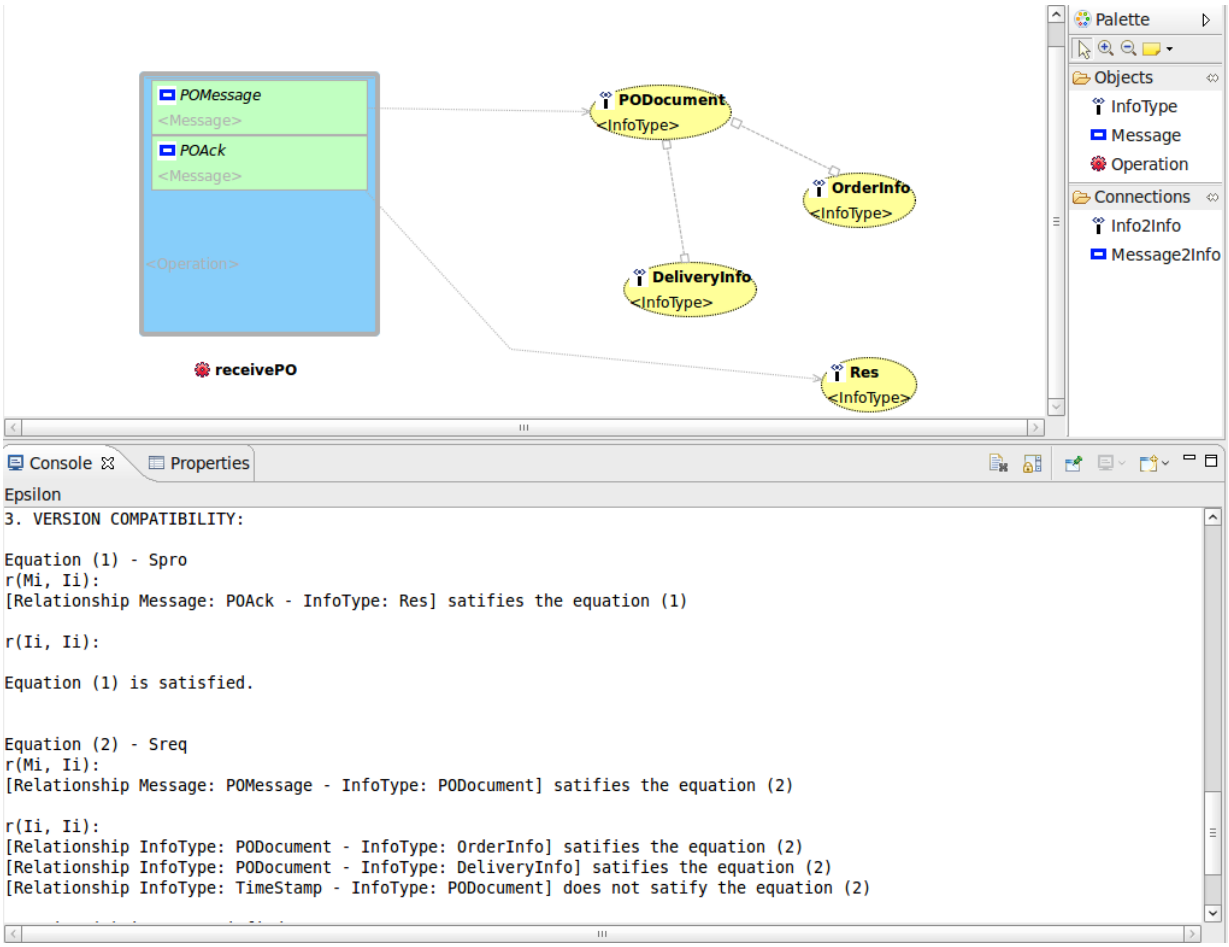


Figure 8.3: SRM prototype – reasoning module

ical analysis that we performed while checking for the T-shaped property in the previous chapters agrees with the algorithmical implementation we presented.

8.2 Validation Experiment

In this section we validate the compatible service evolution model we presented in Chapter 6 by using the SRM prototype. In particular, we present the experimental setup that allowed us to validate our proposal along two dimensions:

- by confirming the theoretical results as produced by the model through the SRM reasoning module, and
- by comparing them with the respective evolutionary experiences using current Web services-supporting technologies.

The first dimension provides us with evidence towards the validity of our theory. More specifically, by automating our theoretical compatible service evolution model in an implementation, we demonstrate that the model is realizable. The second dimension allows us to evaluate the efficacy and applicability of our model. For this purpose we compare it to the compatibility capabilities offered by the dominant Web services description language specifications and their implementation technologies.

In the following we discuss the parameters of the experimental setup, the results of the experiment and their interpretation. These results are then used in the discussion on the realization of service evolution with respect to the dominant Web services standards.

8.2.1 Setup

The validation experiment is decomposed in the following steps:

1. Identification of the most widely accepted Web services description language specifications relevant to our service representation model.
2. Selection of an appropriate Web services deployment environment from the State of the Art to host our experiment.
3. Emulation of the implementation of an evolving service in the deployment environment.
4. Design and analysis of the T-shaped property for a selection of the evolutionary scenarios for the service.
5. Deployment of the different versions produced by the evolutionary scenarios.
6. Development of an SBA client for the initial version of the evolving service.
7. Monitoring of the behavior of the client and the service deployment environment when the client attempts to interact with each service version.
8. Comparison of the results of this process with the T-shaped property analysis.

In Chapter 4 we presented various service representation standards in our effort for developing a service representation model suitable for our compatible service evolution model. As we discussed though, despite the existence of many service description languages like WSOL and OWL-S, the undisputed in terms of acceptance and technological support Web service description standards is the WSDL specification. Looking into suitable deployment environments we opted for the very popular stack of the Apache Axis2 Web services engine[4] (version 1.4.1) hosted in an Apache Tomcat servlet container[5] (version 6.0.14). Both

[4] `http://ws.apache.org/axis2/`
[5] `http://tomcat.apache.org/`

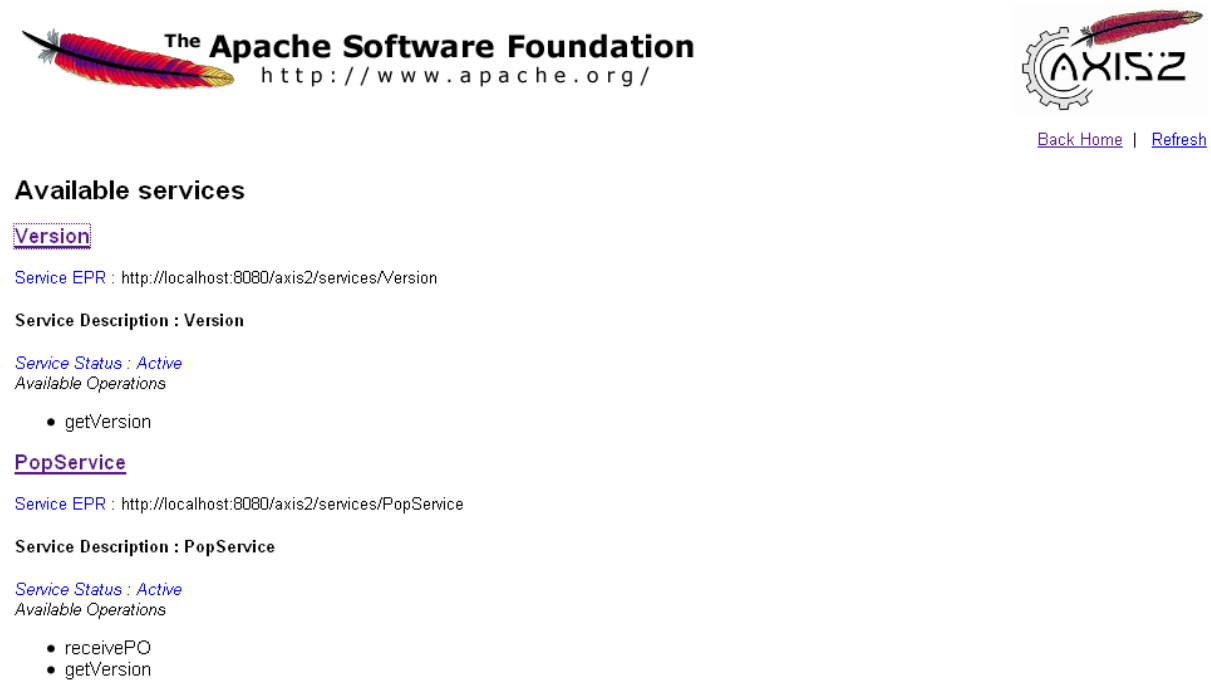


Figure 8.4: PopService deployed in Axis2 service container

solutions are developed by the Apache Software Foundation, are open source, and have been embedded in solutions like the JBoss Application server[6].

For the choice of the emulated service and given the use of the PopService throughout the previous chapters we decided to adopt it for this evaluation process. As a first step we augmented the existing WSDL file of PopService (Listing 3.1 in Chapter 3) with endpoint-specific, protocol-binding information. We then used the WSDL2Java code generation tool[7] in the Axis2 toolkit to generate a skeleton of the service from the initial version of the service, to which we added the necessary business logic for implementing the functionality of the service. We compiled, packaged and deployed the resulting Web service in a Tomcat instance, with the results shown in Fig. 8.4.

Since we need the service to evolve, we applied a similar procedure to the WSDL files for Change Scenarios I to III, also defined in Chapter 3, and created deployable packages of the three versions of the service. In order to simplify the experiment we opted not to deploy them in parallel with the initial version of the service, but to deploy each version on demand. All service versions are sharing the same namespace identifier[8] and their version identifier is retrievable by the `getVersion` operation offered by PopService (Fig. 8.4). Having different namespace identifiers for each version would break the client by default, so the same namespace is used for all service versions.

[6] `http://labs.jboss.com/jbossas/`
[7] `http://ws.apache.org/axis2/tools/1_4_1/CodegenToolReference.html`
[8] `http://fnord.autoinc.com/PurchaseOrderProcessing`

From Chapter 6 we know that applying change sets $\Delta\mathcal{S}_I, \Delta\mathcal{S}_{II}$ and $\Delta\mathcal{S}_{III}$ to the initial version of the PopService $\mathcal{S}_0$ (resulting in service versions $\mathcal{S}_I, \mathcal{S}_{II}$ and $\mathcal{S}_{III}$, respectively) is T-shaped only for $\Delta\mathcal{S}_{II}$. In order to investigate all available aspects of the evolution of PopService we create three more change scenarios by inverting these change scenarios: $\overline{\Delta\mathcal{S}_I}$ is the change set that when applied to $\mathcal{S}_I$ results to the original version of the service: $\mathcal{S}_I \circ \overline{\Delta\mathcal{S}_I} = \mathcal{S}_0$. Similarly, by applying $\overline{\Delta\mathcal{S}_{II}}$ and $\overline{\Delta\mathcal{S}_{III}}$ to $\mathcal{S}_{II}$ and $\mathcal{S}_{III}$, respectively, we revert to the original version $\mathcal{S}_0$. By repeating the analysis performed in Chapter 6 we can deduce that:

- $\overline{\Delta\mathcal{S}_I}$ is T-shaped – it can be easily shown since it results into a more general (in sense of accepted input) service.
- $\overline{\Delta\mathcal{S}_{II}}$ is not T-shaped because of the removal of an interaction path from the interaction protocol.
- $\overline{\Delta\mathcal{S}_{III}}$ is not T-shaped since it requires the removal of the time stamp information from both the input and the output of the service, resulting in the latter case in a more general record in the $\mathcal{S}_{0pro}$ set (which contradicts Definition 7).

In order to confirm these results with the SRM tool we prepared the different versions in the graphical editor incorporated in the prototype. We then ran the reasoning module on each pair of versions in the change scenarios and recorded the results (Table 8.1).

Developing an SBA to act as the client of the service was also achieved by using the code generation tool provided by Axis2. Starting from the WSDL file of the service we generated a skeleton for a simple service client in Java. Based on the provided transformation of the messages data types into classes we wrote a short business logic that invokes the PopService on a standard endpoint with a sample payload and waits for the call back invocation of the service (as in the case of the PopClient service in Chapter 7). The client logs all outgoing and incoming messages and events.

A version of the client was generated for each of the four initial versions required for all change sets:

1. $\mathcal{S}_0$ for $\Delta\mathcal{S}_I, \Delta\mathcal{S}_{II}$, and $\Delta\mathcal{S}_{III}$,
2. $\mathcal{S}_I$ for $\overline{\Delta\mathcal{S}_I}$,
3. $\mathcal{S}_{II}$ for $\overline{\Delta\mathcal{S}_{II}}$, and
4. $\mathcal{S}_{III}$ for $\overline{\Delta\mathcal{S}_{III}}$.

We then proceeded to run each client version against two deployed versions of PopService as defined by each change scenario: the initial version of the PopService (e.g. the client generated based on version $\mathcal{S}_0$ for the change set $\Delta\mathcal{S}_I$, the one based on $\mathcal{S}_I$ for $\overline{\Delta\mathcal{S}_I}$, etc.) and then the resulting version of the change set. During each invocation we were monitoring the server and client logs in order to find out whether the service is invoked successfully and returns the expected acknowledgement message, or whether the service or the client breaks. The results of this process and their analysis follow.

8.2.2 Results & Analysis

Table 8.1 summarizes the findings of our experiment. More specifically:

Change Scenario	Modification	T-shaped	SRM Check	Client/Service breaking
I	$\mathcal{S}_0 \circ \Delta\mathcal{S}_I = \mathcal{S}_I$	No	Fail	Yes (if `DeliveryInfo` is not submitted)
$\overline{I}$	$\mathcal{S}_I \circ \overline{\Delta\mathcal{S}_I} = \mathcal{S}_0$	Yes	Pass	No
II	$\mathcal{S}_0 \circ \Delta\mathcal{S}_{II} = \mathcal{S}_{II}$	Yes	Pass	No
$\overline{II}$	$\mathcal{S}_{II} \circ \overline{\Delta\mathcal{S}_{II}} = \mathcal{S}_0$	No	Fail	No (if using only the asynchronous operation)
III	$\mathcal{S}_0 \circ \Delta\mathcal{S}_{III} = \mathcal{S}_{III}$	No	Fail	Yes
$\overline{III}$	$\mathcal{S}_{III} \circ \overline{\Delta\mathcal{S}_{III}} = \mathcal{S}_0$	No	Fail	Yes

Table 8.1: Experimental validation results

- The SRM reasoning module produces the expected from theory results.
- The service or client break in (almost) all the cases that a non-T-shaped change scenario is applied. The only exception is the case of change scenario $\overline{II}$, i.e. the inversion of the change scenario II, in which the synchronous communication capability of the service is removed. Since the client is not using this capability then it is not affected by it. If the client was using the synchronous communication then it would break.
- All T-shaped change scenarios result in compatible (non-breaking) behavior on the client side – as expected.

The experimental results therefore agree with the theoretical predictions with respect to the compatibility of the evolving service. We can thus conclude that our compatible service evolution model is shown to be realizable. However we can not claim that our theoretical model is validated by all possible experimental cases. In Chapter 6 we discussed that while theoretically consistent and sound, our approach deviates from the empirical proposals by allowing changes that are not covered by the backward preservation guidelines.

More specifically, as shown in Table 6.3, the service compatibility theory we developed allows, in addition to the guidelines in Table 6.1, for the removal of an operation (if it uses the one-way message pattern), the modification of operations (if it respects the covariance and contravariance principles), the modification of data types (to more general in input and more specific in output) and the addition and removal of mandatory data types (for output-type and input-type messages, respectively). While the removal of an operation has been handled by change scenario $\overline{II}$ and the modification of data types by change scenarios I, $\overline{I}$, III and $\overline{III}$, the other patterns have also to be checked.

For this purpose we focused on the addition and removal of mandatory data types – covering in this way also the modification of operations and data types that are essentially a combined addition and removal of the respective record(s). We started from an alternative version of the PopService shown in Listing 8.2 that, as in the case of Change Scenario III, contains a `TimeStamp` in both incoming (`PODocument`) and outgoing (`POAck`) messages.

```
<types>
 <xsd:schema>
  <xsd:complexType name="PODocument">
   <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string"/>
    <xsd:element name="TimeStamp" type="tns:TimeStamp"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="POAck">
   <xsd:element name="TimeStamp" type="tns:TimeStamp"/>
  </xsd:complexType>
  <xsd:simpleType name="TimeStamp">
   <xsd:restriction base="xsd:dateTime"/>
  </xsd:simpleType>
 </xsd:schema>
</types>
```

Listing 8.2: Alternative PopService Message Schema

As per the theory, we added an obligatory data type in the outgoing message (`Comment`) – containing an acknowledgement text from the service provider, and removed the obligatory data type `TimeStamp`, as shown in Listing 8.3. From Table 6.3 we know that this change set is T-shaped. When we replicated the experimental procedure we described in the previous and we deployed the two service versions we found out that the service client broke, contrary to the theoretical prediction. Similar results were observed when `receivePO` operation was modified. These results confirm the empirical guidelines proposed for backward compatibility (Table 6.1) but seem to contradict the compatible service evolution model we developed in the previous.

We deemed therefore necessary to investigate further this discrepancy between theory and practice. Our intuition was that the problem stems from the processing of the XML messages in both service provider and client sides. This belief was further reinforced by Ian Robinson's discussion on breaking changes where he describes a similar situation for a community-designed service [9]:

> The service community in this example is frustrated in its evolution because each consumer implements a form of "hidden" coupling that naively reflects the entirety of the provider contract[9] in the consumer's internal logic. The

[9]By the term contract, the author denotes a service representation in our terminology.

```
<types>
 <xsd:schema>
  <xsd:complexType name="PODocument">
   <xsd:sequence>
    <xsd:element name="OrderInfo" type="xsd:string"/>
    <xsd:element name="DeliveryInfo" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="POAck">
   <xsd:element name="TimeStamp" type="tns:TimeStamp"/>
   <xsd:element name="Comment" type="xsd:String"/>
  </xsd:complexType>
 </xsd:schema>
</types>
```

Listing 8.3: Alternative PopService Message Schema – Change Scenario V

> consumers, through the use of XML Schema validation and static language bindings derived from a document schema, implicitly accept the whole of the provider contract, irrespective of their appetite for processing the component parts.

This discrepancy manifests when either side tries to (unnecessarily) validate the incoming and outgoing messages against a schema that is no longer valid – but not necessarily incompatible. This validation is a substitute for the inability of the enabling technologies to dynamically drop information that they do not understand. In both cases, if both parties could ignore the data types that are not in their message schema (that is, `TimeStamp` for the service provider and `Comment` for the service client) and validating the rest of the message, then the result would be an agreement with the theoretical projection.

Essentially, the service compatibility theory proposed by this work assumes an object-like representation of the services and bases its principles on a technology-agnostic treatment of services. When applied to the "reality" of a low level implementation of a service it is confined by the limitations of the technologies used. A work-around for enabling these types of changes is to intercept the messages and apply to them an appropriate transformation using a technology like XSLT[10] or Schematron[11] as discussed in [9].

Nevertheless, it is our belief that this issue is better handled on the level of service description languages rather by building ad hoc work-arounds. For this purpose in the following we distill the results of this experimental process, and the conclusions from throughout the rest of this work in order to discuss how our proposal can be realized on the level of service standards.

[10] eXtensible Stylesheet Language Transformations (XSLT) Version 2.0 `http://www.w3.org/TR/xslt20/`

[11] `http://www.schematron.com/`

8.3 Realization

In order to evaluate the level of preparedness of service description language specifications for implementing compatible service evolution we surveyed their latest versions and adjunct documents for mechanisms that support evolution. In particular, we referred to the WSDL 2.0 Primer [157], the BPEL 2.0 specification [118] and the WS-Policy 1.5 specification [119]. All the surveyed specifications with the exception of WSDL, do not contain a discussion on versioning that, as we have discussed in Chapter 5, is a requisite for service evolution. The WSDL 2.0 Primer on the other hand simply discusses evolutionary strategies for evolving services in a non-normative manner, incorporating the strategies found in [106].

More specifically, by examining WSDL 2.0, we observe that the language is actually much more restrictive than our approach with respect to service evolution. It concentrates on the guidelines for backward and forward compatibility as presented in Table 6.1. Essentially this means that only additional operations, optional data and new service endpoints or additional wiring protocols are enabling compatible service evolution. The authors of the specification though acknowledge the fact that changes in the message content depend on the type system used to describe them. The weak typing approach taken in the processing of messages (most commonly XML) and the static binding of service and client implementations to WSDL documents leaves little space for improvement given the limitations of existing technologies and standards for Web services.

To surmount these limitations the compatible service evolution model proposes:

1. A uniform model for the representation of message content, interaction protocol and QoS dimensions.

2. A strong typing system coupled with this representation model that allows for more flexibility in what constitutes a compatible change, based on a rigorous theoretical foundation.

3. A versioning approach that complements the theoretical aspects of this work and provides for robustness in unambiguously identifying service versions and recording the historical process of the development of a service.

The above points would require the current WSDL specification working in tandem with BPEL & WS-Policy in order to integrate all aspects of service description into a tightly connected set of documents along the lines of our approach. While both BPEL and WS-Policy can refer to WSDL elements in their documents, the integration of the three languages is quite loose on purpose. This fact, in combination with the weak technological support for the other standards in comparison to WSDL, has dissuaded many service providers from exposing any more information than what is contained in a WSDL document.

As we have already discussed though, WSDL is very limited in the amount of information that it is carrying with respect to the needs of consumers. Providing a tighter integration of the three specifications by allowing to refer to elements of the the other

specifications in a document (and not only to WSDL elements from the other two) would be a definite improvement. A vertically integrated document that combines all three specifications like for example the serialization of an ASD model – or alternatively implementing a generic meta-model for the description of service like the OASIS SOA Reference Architecture [122] – would leverage this effort. This would also require support in terms of service implementation and deployment technologies in order to succeed.

Furthermore, a stronger typing system than the current one has to be used both on the level of XML processing (a flavor of which the WSDL, BPEL and WS-Policy languages are) and on the level of the respective standard specifications. The model of simple XML parsing backed by XML Schema validation currently used in most Web services technologies stifles evolution and creates unnecessary coupling in both service provider and consumer sides. Despite the use of extensibility (as discussed in Chapter 6), it is very difficult to design for compatible service evolution without the possibility of ignoring the parts of a message that are not understood by the message consumer.

We therefore propose that the parsing and validation model should be replaced by the automatic *marshaling* of the messages (that is, the transformation into the respective objects) and the *check for compatibility* on the level of records (using for example Algorithm 1). Static bindings should be replaced by reflection-based bindings to interface classes. These classes are able to accommodate the subtyping of the messages and representations through the use of inheritance (in static languages like Java) or a combination of inheritance and dynamic binding of types (in dynamic languages like Ruby[12]). This would in turn translate into a more suitable for evolution technological foundation for SOA in the form of service containers and middleware that enable the application of our proposal.

Finally, the use of XML namespaces for version identification should be replaced by (or combined with) a more robust versioning mechanism, like the provisioning for version attributes as part of the service description document. While very practical and easy to implement, namespace-based techniques depend exclusively on the service developer to be realized as shown by our validation experiment. This dependence increases the propensity for errors and miscommunication. Furthermore, using a different namespace identifier for each modification unnecessarily breaks the service clients and increases the maintenance cost by introducing additional versions.

As we discussed in Chapter 5 though, namespaces are a very useful mechanism for ensuring that service clients consume only a compatible service version. Used therefore in conjunction with our previous recommendation for a stronger type system and with the versioning strategy already discussed (with major versions signified by new namespace for major revisions and minor revisions subsumed under one version), they can leverage the management of service evolution. Introducing in the language specification versioning attributes at least on the level of the document, would provide a more natural way of handling minor versions that motivates service application developers to plan for multiple service versions.

[12] `http://www.ruby-lang.org/en/`

8.4 Summary

The previous sections discussed the validation of our proposal by means of a proof-of-concept prototype implementation and an experimental procedure using this prototype. This process complements the theoretical aspect of the validation supported by the formal underpinnings of our proposal, and the practical aspect via the use of a running scenario that unifies the demonstration of applicability throughout this work.

The implementation of the Service Representation Modeler (SRM) prototype provided two functionalities that are required for validation purposes: a graphic editor for defining ASD models representing different service versions, and a reasoning module that compares two ASD models and checks them for compatibility using the theory developed in the previous chapters. SRM was developed using the Eclipse framework that provides a wealth of plug-ins for the manipulation of different types of models and the automated generation of graphic editors for the models.

The SRM tool was implemented by describing the structural aspect of the ASD Metamodel in a suitable textual language and its consequent transformation into a data metamodel that Eclipse can manipulate and visualize directly. Based on this facility we developed the graphic editor for defining ASDs. Furthermore, using a model transformation plug-in of Eclipse called Epsilon we unrolled the compatibility checks discussed in Chapter 6 into a fully realized program that performs the compatibility check for two given models. The feasibility of our compatibility evolution model and the validity of the implementation were confirmed in the experiment we performed by showing that the reasoning module replicated the theoretical results produced by pen and paper.

In particular, as part of the experimental validation we emulated the implementation of different versions of the PopService as defined by the change scenarios in Chapter 3 and the inversions of these scenarios (resulting back to the original version). By deploying the different versions into a Web service container and automatically generating a client for each version we aimed at checking whether the client would break in accordance with the T-shaped property. The results confirm the validity of our approach for the set of scenarios we chose. Expanding the procedure to a different scenario though led to some disrepancy between the theoretical and practical findings.

More specifically, it was found that for some cases the client was breaking despite the fact that the change is T-shaped. We investigated further and we discovered that this discrepancy is due to the simple message parsing/validation model based on weak typing used by the majority of services technologies. For that purpose, and in combination with the overall findings of this work, we proposed three significant modifications to the services specification languages: the tighter integration of the specifications for different layers, a marshaling/compatibility checking model based on strong typing to replace the parsing/validation one used currently, and finally, the addition of versioning information in the service description documents to facilitate the management of service versions.

Chapter 9

Conclusions & Future Work

> On those stepping into rivers the same, other and other waters flow.
>
> *attributed to Heraclitus of Ephesus*

> "What's next," these days, is always a cloud, not a single arrow.
>
> *Warren Ellis*

9.1 Summary

Software services are subject to constant change and variation. On one hand, service-orientation increases an organization's agility and decreases the cost of change by minimizing the dependencies between services and allowing them to be recomposed on demand. An organization can only fully realize these benefits, however, if its services are able to evolve independently of one another. On the other hand, services have also to cope with fixing bugs and other errata, dealing with changing requirements, providing desirable variations and performing readjustments to fit their implementation. Such changes can happen at any stage in the service life cycle and they may have an unpredictable effect on the service stakeholders. Being therefore able to control how changes manifest in the service life cycle is essential for both service providers and service consumers.

Towards this goal, this work presented a framework that assists service developers in controlling and managing service changes in a uniform and consistent manner. For this purpose we distinguished between shallow (small-scale, localized) changes and deep (large-scale, cascading) changes and we opted to deal with shallow changes. In particular, we provided service developers the theoretically supported means to deduce and appropriately constrain the effect of changes to a service, so that the changes are shallow. Since, due to the encapsulation of services, changes to the service implementation are transparent to the service consumers and are of concern only when they have an impact on the service interfaces, we focused on changes to the description of services.

While there are enough existing works dealing with the management of change in software there are not many that are concerned with the evolution of services. The vast majority of the existing approaches in the field are either adopting an adaptive methodology, aiming to resolve conflicts and mismatches in the service description as soon as it appears, or they are trying to control service evolution by restricting applied changes to a set of prescribed change patterns. The goal in both cases is to preserve the compatibility of services (both with their consumers and between versions). In the case of the latter category, to which our work belongs, compatibility is usually enforced by a set of empirical and technology-specific rules that dictate which changes are classified as compatible. No theoretical foundation that justifies these rules is provided, and any change in the language of service description will require the modification of these guidelines.

For these reasons we presented in the previous chapters our proposal for a formally-backed compatible service evolution model. This model is based on a technology-agnostic notation for the representation of services in the form of Abstract Service Descriptions (ASDs) that we also introduced. The ASD model comes equipped with mappings to some of the most popular WS-* standards (WSDL, BPEL and WS-Policy) but it is not capable of representing any versioning information. After a survey of existing approaches on service versioning we concluded that the proposed techniques and strategies are sufficient for our purposes. As a result we proceeded to augment the ASD notation accordingly with versioning identification mechanisms so that we can uniquely identify service versions in the development continuum.

Using these results, we defined service compatibility both informally and formally and developed a theory for the compatible evolution of services. As part of the discussion we defined the notion of *T-shaped* changes (that is, resulting in compatible service versions) and we showed how to reason on service versions in order to decide whether their changes are T-shaped or not. Service compatibility was identified as a sufficient condition of ensuring that only shallow changes occur to services. We validated our compatible service evolution model in practice by means of a proof-of-concept prototype implementation and an experimental procedure. Based on the findings of this validation we provided a series of recommendations for the improvement of service description languages towards the direction of service evolution. We also presented an alternative model for managing the evolution of services using bilateral agreements between service providers and consumers, that expanded beyond the notion of T-shaped changes.

The rest of this chapter concludes this work by assessing our findings against the research questions posed in the introduction, presenting the key contributions of our work, evaluating the overall effort and its limitations and briefly discussing future research directions.

9.2 Research Results

Research Question #1

> *What is the State of the Art in service evolution and how is evolution treated in relevant research fields?* What are the techniques, theories and lessons that can be taken from the literature and the industrial practice?

The State of the Art in service evolution has been primarily established in Chapter 2. Relevant works have been classified according to how they approach evolution with respect to compatibility in *corrective* (adaptation-based) and *preventive* (change-restrictive) approaches. A number of approaches that are indifferent to compatibility and aim at supporting evolution without considering shallow or deep changes have also been identified. Due to the purpose of this work these approaches are of lesser interest.

Corrective approaches have been further classified into *service adaptation* and *adapter generation* works. Service adaptation is further distinguished into *interface adaptation* approaches, where the signature of the service is modified to adapt to a new environment, and *composition adaptation* approaches that focus on composite services and attempt to recompose or replace consumed services to achieve the adaptation goal. While adaptation-based techniques are limited by the adaptation scenarios that can be treated automatically, they can leverage the compatible service evolution by ameliorating identified incompatibilities. They can therefore be used in conjunction with the compatible service evolution model discussed in this work to adapt to non-T-shaped changes.

The majority of preventive approaches investigated in this work, from both the industry and the academia has been found to suffer the same weakness. In particular, they all use an implicit notion of (backward) compatibility in order to define what type of changes are allowed to occur to a service. They rely on empirical guidelines for the definition of compatibility, that while widely acceptable from the practitioner's perspective, they lack the validation capability of a formal method and they depend on the specifics of the technology used for representing services (most notably WSDL). This clear need is addressed by this work.

In terms of other relevant fields, and apart from providing the motivation and establishing the context of this work, investigating into software evolution also made clear that a different set of tools than the traditional ones are required for managing the evolution of large distributed systems like a service chain. Versioning techniques from SCM are for example irreplaceable, experiences from versioning Component-Based Systems (CBS) carry essential lessons, and theories from object-orientation can be updated suitably for use in service evolution. They all have to be evaluated however against their applicability in a loosely-coupled, strongly encapsulated environment.

Research Question #2

> *How can evolving services be represented in a uniform across service layers manner?* What are the dominant trends in service interface description and

how do they incorporate service evolution?

Our investigation into service interface description initiatives have turned up with mixed results. WSDL is one of the most popular specifications for vendors and researchers alike and forms the backdrop for all discussions on service representation. It covers though only one aspect of services (the structural) and it does not include the facilities for handling the evolutionary process natively. Due to the heavy reliance on XML it is possible to employ a number of smart techniques for uniquely identifying a service version (namespaces, custom attributes, registry metadata and combinations of the above). Combined with a set of assumptions about the versioning strategies to be followed, these techniques have been proven very successful in the field of service engineering.

Nevertheless these solutions are far from optimal. The limitation of WSDL to the structural layer was attempted to be addressed by the development of the WS-* technological stack but the overabundance of proposed solutions led to a standstill. Furthermore, the lack of backing from the industry and the consequent limited adoption, has sentenced a number of academic standards for service representation to limited acceptance. For these reasons, in this work we opted to align our research with initiatives like the OASIS SOA Reference Architecture and define a technology-agnostic formal model of service representation in the form of ASDs that covers all service aspects instead of "inventing" yet another service description language.

Furthermore, we observed that the versioning techniques that have been deployed so far have always been circumstantial adjuncts to the service signatures and not an integrated aspect of the service life cycle. It remains the responsibility of the service developers and managers to understand the assumptions used and to process the versioning information. Versioning-supporting mechanisms must therefore be incorporated into the service description languages specifications in order to become a standard in the development and deployment of services.

Research Question #3

> *What exactly constitutes service compatibility?* A theoretical and practical definition of compatibility in the context of services is required to allow the definition of when evolving services are compatible.

Compatibility is one of those terms that has been so overloaded with definitions that it is too difficult to choose the most appropriate one for our purposes. For that reason we opted to integrate two different aspects of compatibility that one encounters in the literature. More specifically, we differentiate between *vertical* compatibility, referring to the property of versions to be interchangeable under controlled conditions, and *horizontal* compatibility, denoting the interoperability of two services, one acting as a provider and the other as a consumer. Since the vertical aspect can be seen as the property of preservation of the horizontal one (and vice versa), we defined the concept of *T-shaped compatibility* combining both aspects.

During the interaction with another service or client, a service acts as both a language (in terms of messages) producer and consumer. That leads to the traditional decomposition of compatibility into *forward* (with respect to the evolution of the message producer) and *backward* compatibility (with respect to the evolution of the message consumer). Full compatibility is the combination of both forward and backward compatibility. In order therefore to give a clear definition of service compatibility we have to incorporate both aspects of compatibility (vertical/horizontal and forward/backward). As a result, in Chapter 6 we defined *service* compatibility as the satisfaction of the *covariance* criterion for the part of the service that acts as language producer (meaning that the output can only be specialized) and the *contravariance* criterion for the part that acts as language consumer (i.e. the input can only be generalized). Full service compatibility is thus the satisfaction of both criteria.

Research Question #4

> *What are the conditions that enable compatible service evolution?* How does the definition of service compatibility interact with the evolution of services? How is it possible to constrain the type of changes to a service to a set of compatibility-preserving ones? What are the benefits of this evolutionary model with respect to the State of the Art?

Equipped with our definition of service compatibility we can equate compatible service evolution with the satisfaction of the criteria set by the definition. Checking for compatible service evolution is reduced to algorithmically checking whether the covariance and contravariance properties are respected during the evolution of the service. *Change sets* (groups of primitive changes that are applied together to a service) can thus be categorized to *T-shaped* (preserving service compatibility) and non-T-shaped. Our approach is not only able to replicate the (backward) compatibility preservation guidelines that are ubiquitous in other approaches but it can also produce much more refined results in reasoning about service evolution.

Constraining the types of changes to compatibility-preserving ones requires the ability to reason on versions of service descriptions. For this purpose we updated the classic *type theory* from object-oriented languages by fitting it into our service representation model and extending it to cover all three aspects of services. More specifically, of particular interest for service compatibility is the subtyping relation between records, signifying their specialization and generalization. While it is straightforward to define this relation for the structural aspect, we had to resort to existing approaches that define equivalent relations for the other layers (i.e. behavioral and non-functional).

A version of type theory suitable for service representations was the outcome of this process. This theory allows us to compare two versions of services on the basis of their constituent records and conclude whether one can conditionally replace the other. This replacement, as driven by the definition of service compatibility, is only allowed towards more general input and more strict output. In that sense, our model of compatible service

evolution is essentially an application of Postel's Law[1]:

> Be conservative in what you do; be liberal in what you accept from others.

Research Question #5

> *Is service compatibility equivalent to shallow changes? Are there alternative models of shallow changes outside of the service compatibility one?* Can the restrictions to the allowed changes to a service be relaxed? What are the benefits and disadvantages of such a solution?

In Chapter 7 we have demonstrated by construction that there are indeed alternative models for ensuring that the changes occurring to a service are shallow. For that purpose we used the concept of *service contracts*, i.e. bilateral agreements between service providers and clients in the form of intermediate service representations. For the purpose of forming and configuring the contracts we used the subtyping relation we discussed in the previous chapter and a similar reasoning as the one for checking for compatibility. Service evolution using contracts is only constrained by the capability of both parties to respect the contract between them while manifesting changes.

The compatible service evolution model we presented in Chapter 6 required no special effort for managing the evolution of services beyond that of being able to reason on change sets. The introduction of service contracts however requires additional provisioning. In particular, the infrastructure for and the capability of forming, configuring and exchanging and maintaining contracts has to be put into place at possibly significant expense. Furthermore, service contracts are not a magic bullet for unbound service evolution. Basically, the service becomes implicitly coupled to its clients since its evolution depends on their capacity to withstand changes. Flexibility is therefore traded for coupling and overhead.

Research Question #6

> *How can the proposed solution be validated practically? Can the theoretical results be replicated by a prototype? What are the limitations of the proposed solution?* A proof-of-concept implementation is required in order to demonstrate the realization of the solution. Furthermore, an evaluation of its realization with respect to existing technologies and standards is necessary.

Chapter 8 discusses the validation of the proposed compatible service evolution model in practice. This validation is performed by verifying the theoretical findings through an experimental procedure using the prototype implementation of the Service Representation Modeler (SRM) tool. More specifically, the SRM prototype, developed using the Eclipse platform, provides two key functionalities for validation purposes: a graphic editor for defining service representation models and a reasoning module that implements our service

[1] `http://en.wikipedia.org/wiki/Robustness_principle`

compatibility theory. Based on this prototype we defined a procedure for checking the validity of our proposal using the change scenarios used throughout the rest of this work.

For this purpose we created a mock implementation of each service version in the change scenario and we deployed them in a Web services container. We then automatically generated a simple service client for each of them and used the client to check if it "breaks" by invoking different service versions. The results of this procedure confirmed the theoretical analysis of service compatibility performed both on paper and by the SRM prototype.

We also expanded our validation procedure to modifications that are not covered by the State of the Art on service evolution, but which they should be (according to our theoretical model and the prototype implementation). The results of this secondary validation showed a discrepancy between the theoretical prediction and the actual behavior of the service client with respect to compatibility. After the required investigation was performed, it was deduced that this discrepancy is due to the technological limitations imposed by the implementation of the major Web services description language specifications. In particular, the inability of the message processing mechanisms to drop information that is not contained in the original message schema for application safety purposes is a serious hindrance to the service compatibility theory achieving its full potential. For that purpose we provided a set of specific recommendations about how the major service specification languages can be modified to accommodate service evolution in a more natural and efficient way.

9.3 Contributions

The results of this work address the need for a comprehensive, theoretically-supported model for the management of service evolution. A set of theories and models that unify different aspects of services into a common reference framework for the representation, versioning and evolution of services has been developed for this purpose. This framework pushes forward and redefines the State of the Art in service evolution. It achieves this by replicating and formally validating the empirical findings and best practices for service evolution. At the same time it outlines a number of possibilities for service evolution that are not currently covered by existing standards and technologies.

The major results of this work with respect to the State of the Art in service evolution and service science are:

A technology-agnostic uniform formal model for the representation of service interfaces and their different versions.

In Chapter 4 we presented a service representation model based on *Abstract Service Descriptions (ASDs)*. The developed service representation model seamlessly integrates the different aspects of services (structural, behavioral and non-functional) into one model. The ASD Meta-model (Fig. 4.1) aggregates the concepts from the meta-models of WSDL, BPEL, WS-Policy and the other representation initiatives discussed in the chapter like

the OASIS Reference Architecture, the CDBI-SAE Meta Model for SOA and the SeCSE facet-based specification approach. The ASD representation model is not meant as a new service description language but as a formalism for the representation of services.

The formalization of the ASD notation is founded on the structural layer, extending the semantics of the UML class diagram notation for describing the dependencies between the elements of this layer. This foundation is then extended accordingly with a mapping to behavioral contracts for the behavioral layer, and with the capacity for representing QoS characteristics in the form of ordinal QoS dimensions. The formal foundation enables the use of type theory for reasoning on the evolution of ASDs and forms the basis for the discussion on all following chapters. It also allows the definition of ASD *consistency*, denoting the well-formedness of ASDs according to a set of *invariants* (namely: validity with respect to the meta-model, reachability of elements and property preservation).

An example of this relation of the ASD model with the rest of this work can be demonstrated by its use in recording the historical aspect of service evolution. By introducing versioning information to the level of ASD records (elements and relationships) we are able to uniquely identify particular instances of records in the development continuum. Since ASDs are defined as the sets of records that they contain, versions of services manifest as *versioned ASDs*. For each service version it is thus possible to track down the history of both the service (in terms of the sets of changes that were applied to it) and its constituent records (by going through their individual versioning histories).

A theory and model for the compatible evolution of services.

The major contribution of this work is the identification and formalization of the conditions under which services can evolve while preserving their compatibility. The conditions are expressed as permitted sets of changes (being T-shaped in our nomenclature) that can occur safely to a service. T-shaped changes respect the definition of *service compatibility* we presented in Chapter 6 as the combination of the properties of covariance and contravariance.

Algorithm 1 presents a straightforward compatibility checking algorithm for evaluating these properties. The reasoning required is performed on the basis of versioned ASDs and uses the updated and extended subtyping relation that we developed for this purpose (Definitions 9, 10 and 12). We demonstrated the applicability of the compatible evolution model by checking the change scenarios to the PopService that we described in Chapter 3. Based on the results of this checking we proposed different evolutionary scenarios for each case.

Furthermore, the proposed model has been compared to the relevant approaches in preventive evolution. Table 6.3 compares for that purpose the set of T-shaped patterns of change that correspond to (and go beyond) the compatibility preservation guidelines that are used in the State of the Art. It is thus shown that our theory replicates the empirical guidelines, while at the same time it allows for types of changes (under given conditions) that are too specific to be included in the guidelines. Through this procedure we also showed that the proposed model can be used to *generate* possible shallow changes

in addition to checking to them for compatibility.

In addition, the proposed model was evaluated with respect to its *novelty* and *relevance* to service orientation. In terms of novelty, it was shown that it integrates a set of existing and widely-adopted theories like type theory into a common framework of reference for the compatible evolution of services. The existence of an underlying meta-model and the coarse granularity of the ASD representation model differentiates our proposal from similar works in the component orientation domain in terms of its feasibility and efficiency. The focus on shallow changes is further justified by the leveraging of the SOA-specific properties supported by the model, like the document-based communication, the loose coupling, the coarse interfaces and the context-free invocation of services.

A contract-based model of service interaction and evolution.

In Chapter 7 we reused the tools we developed for managing the compatible evolution of services in order to provide an alternative model of service interaction and evolution in the form of *service contracts*. Service contracts are introduced between service providers and consumers as bilateral agreements that specify explicitly the expectations and obligations of both parties. The formation of the contract can be handled automatically by using the subtyping relation we have already defined on the combination of the service provider's and client's representation of the service (in ASD notation). Human input is required only for the configuration of the contract terms, but this can also be avoided by a priori defining appropriate configuration policies.

Based on service contracts, an alternative evolution model was proposed that expands the permitted set of changes and provides more flexibility in evolution. Using a similar reasoning as in the case of compatible service evolution we showed that services in either side of the interaction (that is, both providers and clients) can evolve beyond the T-shaped property, while still being shallow. The contract between them can also be subject to change as a result of the change to one party, assuming that it does not disrupt the other side. We provide the reasoning mechanisms for all the necessary checks by remixing and applying the ASD formalism capabilities and the subtyping relation on ASDs. Nevertheless this flexibility is gained at the expense of increased coupling, additional governance requirements and communication overhead.

An identification of the limitations of existing specifications and technologies with respect to service evolution, and a proposal for their improvement.

The dominant language specifications for Web services description were evaluated on the basis of their support of compatible service evolution using the findings of this research as a benchmark. As a result, a proposal for their improvement was put forward. In particular, it was concluded that the existing languages lack support for compatible service evolution in three important dimensions: the *loose coupling* between the languages for the description

of the different service aspects, the *weak typing system* used for the validation of incoming messages and the dependence on a too *coarse-grained mechanism* for versioning in the form of version identifiers in (XML) document namespaces.

Towards the direction of improving them, we provide a series of recommendations for each one of these dimensions. More specifically, we propose the tighter integration of the service description languages (both in the level of specification and implementation) by allowing constructs from other languages to appear in the document of a language (and not only of WSDL constructs appearing in BPEL documents for example). A new type of service description document incorporating the three aspects of services (structural, behavioral and non-functional) as discussed in the previous could help, provided that it has the appropriate technological (and of course political) support from the industry.

With respect to the model of processing incoming messages, we propose the substitution of the unnecessarily strict model of parsing and validation of the messages against the original message schema (before marshaling – that is, translating – them into objects) by a marshaling/compatibility checking one. This will enable for further flexibility than currently provided for service evolution, and will allow the realization of our service compatibility theory on the level of specification and implementation technologies. In addition, the presence of a version identifier that is understood by the underlying technological solutions, in conjunction to the namespace identifier mechanism for major revisions, would enable a more fine-grained management of service versions.

9.4 Evaluation & Limitations

As we discussed in Chapter 1, providing service developers with the means to control the evolution of services belongs conceptually to design science. As such, the evaluation of this work is performed along the lines of requirements for effective design science research. For this purpose we use the guidelines proposed in [14]. More specifically:

Problem Relevance As previously discussed, SOA increases an organization's agility by encapsulating business functions in reusable services. An organization can only fully realize the benefits of SOA, however, if its SOA instantiation enables services to evolve independently of one another. The existing works on service evolution management are based on empirical findings and best practices, usually relying on the specifics of the technologies used to achieve their purposes. This fact makes these approaches vulnerable to technological shifts. As a remedy to this situation we propose to base service evolution management on a theoretical foundation that enables a technology-agnostic approach of the issue. This is a clear and important need that this work is geared to address by focusing on controlling service changes so that they are shallow. Deep changes, while being also very interesting and relevant are out of the scope of this work.

Design as an Artifact In summary, this work proposed a series of viable artifacts in the form of:

- a formal model for the representation of services (in the form of the Abstract Service Description (ASD) model) and service versions (through versioned ASDs),
- a method for evolving services in a compatible manner with the compatible service evolution model and the T-shaped changes,
- an instantiation of this method in the form of a prototype, the SRM tool, and
- an alternative method for the compatible evolution of services using service contracts between service providers and consumers.

Each of these artifacts was defined in a formal setting and explained using the industrial case presented in Chapter 3.

Research Contributions The research discussed in this work contributes both with the design artifacts themselves (presented above) and with the developed formal foundations in the form of a theory of service compatibility. The contributions of this work have been presented exhaustively in Section 9.3.

Design Evaluation The utility, quality and efficacy of the proposed methods for the compatible evolution of services has been evaluated using experimental and descriptive methods. More specifically:

- Due to the innovative nature of our approach which combines theoretical with empirical aspects we opted to evaluate our approach using the scenarios driven from the industrial case of the PopService. In Chapters 6 and 7 we demonstrated the impact of changes with varying complexity to the consumers of PopService, describing how to avert consumer disruption through the application of the service compatibility theory.
- The proposed method of controlling service compatibility was compared and contrasted with existing approaches in Chapter 6. We concluded that our approach replicates and refines the results of the existing approaches.
- In Chapter 8 we designed and executed an experiment to validate our (theoretical) findings in a simulated environment of evolving service providers and consumers. The prototype focused on the structural layer of services, working exclusively with WSDL descriptions. The experiment showed some discrepancy between theory and practice which was investigated further and explained accordingly.

Research Rigor The methods for preserving service compatibility produced by this work are based on type theory. Type theory has a rigorous mathematical foundation that has been validated both theoretically and empirically through many years of research. This powerful foundation allowed us to develop a proof-of-concept prototype that demonstrated

the realization of our proposal. Nevertheless we preferred to put emphasis on the applicability of our approach and did not pursue to prove the mathematical rigor of the extensions to type theory we proposed. In particular, as we will discuss in the future work section, we need to prove the *closure* and *completeness* properties of our proposal.

Design as a Search Process The produced design artifacts update and expand tested and tried solutions for the services environment. The ASD service representation model abstracts information from the meta-models of widely accepted WS-* languages. The versioned ASDs discussed in Chapter 5 build on the methodology and terminology championed by the SCM community, updating it accordingly for the ASD model. The compatible service evolution model combines type theory with the ASD model in order to show how services can evolve in a compatible manner. The work on service contracts remixes the service compatibility theory and provides an innovative approach to contracting that remains faithful to the original ideas about binding agreements between interacting parties. The results of this research therefore progress significantly models and methods from the existing knowledge base to accommodate the requirements of the SOA field.

Communication of the Research The material in the previous chapters provides information that while interesting for both technical and managerial audiences, they are more suitable for the former type. The discussion revolves around a rigorous mathematical framework which provides explicit mappings to popular services standards like WSDL, BPEL and WS-Policy. Furthermore, the theoretical foundations are presented on the basis of the scenarios drawn from the industrial case scenario and it is easy to demonstrate their applicability. The introductory chapter motivates the need for a comprehensive service evolution management solution and rationalizes the choices made in achieving this solution. In addition, a discussion on the service providers' and consumers' benefits and costs of our proposals takes place in Chapters 6 and 7. Nevertheless, a concrete method for estimating the cost of a change is unavailable and the discussion remains on the level of informed arguments. Such a method, despite being out of scope for this work, would be of further interest for managerial purposes.

9.5 Future Work

As previously discussed, due to the emphasis on the applicability and realization of our proposed solution we have relied on existing theories to ensure the rigor of our research. Nevertheless, since we extended these theories in significant ways in the previous chapters we also have to demonstrate that our extensions are equally rigorous. In particular, in order to fully confirm our hypothesis equating shallow changes with compatibility it is required of us to prove that the set of shallow changes is *closed* under compatibility-preserving change sets. Furthermore we also need to prove that our approach is *complete* in the mathematical sense by showing that it can generate all possible compatible change sets. Fragments of

these proofs are scattered throughout the discussion of Chapter 6 but they have to be collected and formally put together in the immediate future.

The next important action item is the incorporation to the SRM prototype of behavioral and non-functional layer capabilities and the implementation of all envisioned functionalities. More specifically, the prototype, as described in Chapter 8 is able to reason only on the structural aspect of service description. While this functionality was sufficient for the purposes of validating our theoretical results, the capability of reasoning on the other layers is essential for the full instantiation of our proposal. Furthermore, the option to import ASD models directly from WSDL, BPEL and WS-Policy documents using the mappings discussed in Chapter 4 and to visualize the results of the compatibility check will also be added to the SRM capabilities.

These changes will allow us to provide service developers with a comprehensive toolset for controlling the different aspects of service evolution. Towards this goal, the SRM prototype can be significantly augmented by connecting it with a suitable revision control system which will manage the versioning of service descriptions as discussed in Chapter 5. The integration of the reasoning capabilities of SRM with the version recording and communication facilities like those provided for example by the VRESCo environment [102] would create a complete service evolution management solution.

Furthermore, we believe that an observational evaluation of our research results would be appropriate in addition to the descriptive and experimental evaluation already performed. The application of the SRM prototype in one or more appropriate industrial case studies would allow us to draw useful conclusions about the efficacy of both the prototype and our work in general. Of course this process would also allow us to further improve and extend our solution. An organization with a diverse and evolving service portfolio would be the ideal testing grounds for our proposal.

Finally, the evaluation of our proposal on service contracts was limited to the descriptive type given the absence of a suitable implementation. In order therefore to provide a more rigorous evaluation and improve the quality of our work we have to provide a prototype of the necessary infrastructure for service contracts. In contrast though to the SRM prototype, and as discussed in Chapter 7, a simple toolset is not sufficient for these purposes. We have to provide service providers and consumers with a complete solution that is able to *a)* form contracts between parties given their descriptions, *b)* perform all the contract compliance and compatibility checks, and *c)* store and communicate the different events in the contract life cycle (creation of a new contract, update or decommission of an existing contract). While many components of the SRM prototype can be reused in building such system, its complexity and scale would call for a separate research effort.

Applications & Extensions

Apart from the open issues of this research effort that need to be resolved in the short- to medium-term, we also plan to work on applying the design artifacts produced towards different directions. In the following we discuss some of the possible applications and extensions that we have identified.

The ongoing paradigm shift towards a complete transition to Software as a Service (SaaS), fueled by the cloud computing initiatives [170] provides opportunities for applying and fine-tuning our proposal for controlling service evolution. The transformation of applications into services and the abstraction from their supporting hardware and systems using the cloud metaphor enables and justifies one of the basic assumptions of this work: that everything can be described a service. Applying the developed solutions for controlling the evolution of cloud-enabled services can have a profound impact on the adoption of cloud computing in environments of high volatility like today's enterprises.

Furthermore, the compatible service evolution model proposed is not necessarily limited to the Web services approach to service orientation. The theories and models developed could also be applicable to RESTful services [171], that is, services that are built using the REpresentational State Transfer (REST) paradigm [172]. Due to the lack of an interface description language for REST however, RESTful services lack a common representation meta-model that would allow a direct application of our approach to them. Nevertheless, in conjunction with expressing their interfaces in WSDL 2.0 as discussed in [173], we could investigate the extensions of our work in that direction.

Another interesting avenue of future research is the application of our service compatibility theory to the area of service discovery [174]. Discovering an appropriate service becomes essential, and at the same time more complex, with the increase of the number of available services and the availability of different versions of the same service. Similarly to our approach for forming contracts between service providers and consumers, we can drive the discovery and selection of service providers using service compatibility. Generally speaking, given two Web services, the similarity between their interfaces depends on the set of changes to be performed on the interface of the first service to obtain the second one. Changes could be the introduction of new functionalities or modification/deletion of existing ones, but only some of them are able to maintain the compatibility between two versions. Based on this kind of information, we can extend a current Web services matchmaker, named URBE [175], by considering the compatibility among service interfaces while calculating the similarity degree for the matchmaking.

One of the key points for extending our work further is the identified synergy with adaptation-based works. More specifically, as discussed in Chapter 6, the presented approach belongs to the preventive evolution works, constraining changes to a service to the T-shaped ones in order to preserve compatibility. There are however a number of non-T-shaped changes that can be converted into shallow through the use of an adapter to "mask" the effects of the change. In this context, our theory of compatible service evolution can be used for the initiation and the termination of the adapter generation procedure.

In particular, the compatibility checking algorithm (Algorithm 1 in Chapter 6) can be modified in two points. First it should not terminate whenever it finds a record that violates the compatibility check but it should add the record to a separate set. Second, an additional step can be incorporated at the end of the algorithm that checks whether the set of violating records is empty or not. In the latter case an adapter generation subroutine can be called with the two versions and the identified incompatibilities as input. If an adapter can be automatically generated given this input then the algorithm is to be run

again, checking the adapted version against the original one for compatibility. Otherwise, the change set is to be categorized as deep.

Deep Changes

The management of deep changes is another possible extension point for our work. As discussed in Chapter 1, for deep changes it is not sufficient to look only at the structural, behavioral and non-functional but also at policy-induced and operational efficiency changes. Service compatibility has to be connected with models for service governance and compliance in order to guarantee a holistic approach in service evolution management.

More specifically, in [7] Papazoglou introduces the notion of a *change-oriented life cycle* methodology that addresses the problems of modification and alignment of business processes expressed as services within a service network [116]. The life cycle provides a sound foundation for deep service changes in an orderly fashion that allows service to be appropriately reconfigured, aligned and controlled while changes occur. It also provides common tools to reduce cost, minimize risk exposure and improve development agility. It helps organizations ensure that the right versions of the right processes are available at all times, and that they can provide an audit trail of changes across the service life cycle to prevent application failures and help meet increasingly stringent regulatory requirements.

Furthermore, in [176] the authors also discuss deep change management, which in that context entails the timely detection, propagation and reaction to both internal and external changes. The approach is based on using agents – background processes that monitor the participant service for relevant changes and notify entities concerned with the change. While in [176] a bottom-up perspective is adopted, assuming that a change is initiated by an individual service and then propagated to the enterprises that use the service, [177] focuses on top-down changes that are initiated by an entire organization and triggered by new business requirements. Change is handled by a formal model specifying the changes required by one or more services, and by a change reaction component that applies the specified change in a consistent manner. The two approaches are combined in [12], where a *Web Services Management System (WSMS)* architecture, similar conceptually to DBMS, is discussed. Deep change management in this case is one of the key components enabling service management.

Such approaches provide solutions for the organizational issues of change that this work has not dealt with. In combination with the compatible service evolution model presented in the previous chapters, and the extension of our work to include service adaptation (as discussed above), we can take the next step in the effort for developing a solution for the holistic and efficient management of service evolution. By bringing these three aspects together, we will be able to provide service stakeholders with the means to track the effect of proposed and applied changes, estimate the benefits of re-shuffling their service portfolio and decide the fate of existing and new services. In the context of service evolution management, this work can therefore be perceived as the stepping stone on the road for harnessing change.

Appendix A

Acronyms List

ASD Abstract Service Description

AOP Aspect Oriented Programming

BPEL Business Process Execution Language

BPMN Business Process Model and Notation

CCS Calculus of Communicating Systems

CEF Compatible Evolution Framework

CBD Component-Based Development

CBS Component-Based Systems

COTS Commercial-Off-The-Shelf

EMF Eclipse Modeling Framework

FSM Finite State Machine

GMF Graphical Modeling Framework

IDL Interface Definition Language

IT Information Technology

O/O Object-Oriented

OMG Object Management Group

QoS Quality of Service

QRM Quality Reference Model

REST REpresentational State Transfer

SaaS Software as a Service

SBA Service-Based Application

SCM Software Configuration Management

SCOR Supply Chain Operations Reference

SLA Service Level Agreement

SOA Service-Oriented Architecture

SOC Service-Oriented Computing

SRM Service Representation Modeler

TAG Technical Architecture Group

UML Unified Modeling Language

VID Version IDentifier

WSDL Web Services Description Language

WSOL Web Service Offerings Language

XML eXtensible Markup Language

XMI XML Metadata Interchange

Bibliography

[1] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed. Addison-Wesley Professional, Aug. 1995.

[2] M. Fowler, "The new methodology," *Wuhan University Journal of Natural Sciences*, vol. 6, no. 1, pp. 12–24, Mar. 2001. [Online]. Available: http://martinfowler.com/articles/newMethodology.html

[3] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279–287.

[4] M. P. Papazoglou and D. Georgakopoulos, "Introduction to Service-Oriented computing," pp. 24–28, 2003.

[5] M. P. Papazoglou and W. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007.

[6] C. L. Nehaniv, J. Hewitt, B. Christianson, and P. Wernick, "What software evolution and biological evolution don't have in common," in *Proceedings of the Second International IEEE Workshop on Software Evolvability*. IEEE Computer Society, 2006, pp. 58–65.

[7] M. P. Papazoglou, "The challenges of service evolution," in *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. Springer-Verlag, 2008, pp. 1–15.

[8] D. Sprott, "Business flexibility through SOA," CBDI Forum, CBDI Paper, 2005. [Online]. Available: ftp://ftp.software.ibm.com/software/soa/pdf/CBDIWhitepaperBusinessFlexibilityThroughSOA.pdf

[9] I. Robinson, "Consumer-Driven contracts: A service evolution pattern," in *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf, Mar. 2008, pp. 101–120. [Online]. Available: http://martinfowler.com/articles/consumerDrivenContracts.html

[10] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, pp. 38–45, Nov. 2007.

[11] D. Orchard, "Extending and versioning languages: Compatibility strategies [Editorial draft]," Sep. 2008. [Online]. Available: http://www.w3.org/2001/tag/doc/versioning-compatibility-strategies

[12] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and managing web services: issues, solutions, and directions," *VLDB Journal*, vol. 17, no. 3, pp. 537–572, May 2008.

[13] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou, "Managing the evolution of service specifications," in *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. Springer-Verlag, 2008, pp. 359–374.

[14] A. Hevner, S. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 105, 75, 2004.

[15] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decis. Support Syst.*, vol. 15, no. 4, pp. 251–266, 1995.

[16] G. Canfora, "Software evolution in the era of software services," in *Proceedings of the Principles of Software Evolution, 7th International Workshop*. IEEE Computer Society, 2004, pp. 9–18.

[17] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[18] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.

[19] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*. San Francisco, California, United States: IEEE Computer Society Press, 1976, pp. 492–497.

[20] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[21] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*. Springer-Verlag, 1996, pp. 108–124.

[22] M. M. Lehman and J. F. Ramil, "Software evolution - background, theory, practice," *Information Processing Letters*, vol. 88, no. 1-2, pp. 33–44, Oct. 2003.

[23] R. T. Mittermeir, "Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Vienna, Austria: ACM, 2001, pp. 114–121.

[24] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. Limerick, Ireland: ACM, 2000, pp. 73–87.

[25] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[26] R. Arnold and S. Bohner, "Impact analysis-Towards a framework for comparison," in *Proceedings of the Conference on Software Maintenance*, 1993, pp. 292–301.

[27] V. Rajlich, "A model for change propagation based on graph rewriting," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1997, pp. 84–91.

[28] A. I. Antón and C. Potts, "Functional paleontology: The evolution of User-Visible system services," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 151–166, 2003.

[29] W. F. Tichy, "Tools for software configuration management," in *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, Jan. 1988, pp. 1–20.

[30] J. Estublier, "Software configuration management: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. Limerick, Ireland: ACM, 2000, pp. 279–289.

[31] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of software engineering research on the practice of software configuration management," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 4, pp. 383–430, 2005.

[32] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232–282, 1998.

[33] K. Narayanaswamy and W. Scacchi, "Maintaining configurations of evolving software systems," *IEEE Trans. Softw. Eng.*, vol. 13, no. 3, pp. 324–334, 1987.

[34] S. Sowrirajan and A. van der Hoek, "Managing the evolution of distributed and interrelated components," in *Software Configuration Management*, 2003, pp. 243–247.

[35] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, Dec. 1997.

[36] A. Brown and K. Wallnau, "The current state of CBSE," *Software, IEEE*, vol. 15, no. 5, pp. 37–46, 1998.

[37] A. Elfatatry, "Dealing with change: components versus services," *Commun. ACM*, vol. 50, no. 8, pp. 35–39, 2007.

[38] A. Stuckenholz, "Component evolution and versioning state of the art," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 1, p. 7, 2005.

[39] M. P. Papazoglou, *Web Services: Principles and Technology*, ser. Prentice Hall. Prentice Hall, Jul. 2007.

[40] M. Lehman and J. Ramil, "Software evolution in the age of component-based software engineering," *Software, IEE Proceedings* -, vol. 147, no. 6, pp. 249–255, 2000.

[41] D. Reifer, V. Basili, B. Boehm, and B. Clark, "Eight lessons learned during COTS-based systems maintenance," *Software, IEEE*, vol. 20, no. 5, pp. 94–96, 2003.

[42] J. C. Seco and L. Caires, "A basic model of typed components," in *Proceedings of the 14th European Conference on Object-Oriented Programming*. Springer-Verlag, 2000, pp. 108–128.

[43] P. Brada, "Component revision identification based on IDL/ADL component specification," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 297–298, 2001.

[44] M. Zenger, "Type-Safe Prototype-Based component evolution," in *Proceedings of the 16th European Conference on Object-Oriented Programming*. Springer-Verlag, 2002, pp. 470–497.

[45] J. Banerjee, W. Kim, H. Kim, and H. F. Korth, "Semantics and implementation of schema evolution in object-oriented databases." New York, NY, USA: ACM Press, 1987, pp. 311–322.

[46] R. Zicari, "A framework for schema updates in an Object-Oriented database system." IEEE Computer Society, 1991, pp. 2–13.

[47] R. Bretl, D. Maier, A. Otis, D. J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams, "The GemStone data management system." in *Object-Oriented Concepts, Databases, and Applications*. ACM Press and Addison-Wesley, 1989, pp. 283–308.

[48] D. Edmond, A. Bouguettaya, and B. Benatallah, "Formal correctness procedures for Object-Oriented databases," in *Proceedings of the 9th Australasian Database Conference*, 1998.

[49] W. Kim and H. Chou, "Versions of schema for Object-Oriented databases," in *Proceedings of the 14th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1988, pp. 148–159.

[50] J. Andany, M. Léonard, and C. Palisser, "Management of schema evolution in databases," in *Proceedings of the 17th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1991, pp. 161–170.

[51] S. Monk and I. Sommerville, "Schema evolution in OODBs using class versioning," *SIGMOD Rec.*, vol. 22, no. 3, pp. 16–22, 1993.

[52] F. Grandi and F. Mandreoli, "A formal model for temporal schema versioning in object-oriented databases," *Data Knowl. Eng.*, vol. 46, no. 2, pp. 123–167, 2003.

[53] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds., *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[54] J. Estublier and M. Ahmed-Nacer, "Schema evolution in software engineering databases : A new approach in adele environment," *Computers and artificial intelligence*, vol. 19, no. 2, pp. 183–203, 2000.

[55] E. Bertino and L. Martino, *Object-Oriented Database Systems: Concepts and Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[56] S. B. Zdonik, "Object-Oriented type evolution," F. Bancilhon and P. Buneman, Eds. ACM Press / Addison-Wesley, 1987, pp. 277–288.

[57] R. J. Peters and M. T. Özsu, "An axiomatic model of dynamic schema evolution in objectbase systems," *ACM Transactions on Database Systems*, vol. 22, no. 1, pp. 75–114, 1997.

[58] E. Bertino, G. Guerrini, and L. Rusca, "Object evolution in object databases," in *Dynamic Worlds*, ser. Applied Logic Series. Springer-Verlag, 1999, vol. 12, pp. 219–246.

[59] M. P. Papazoglou and B. J. Krämer, "A database model for object dynamics," pp. 073–096, 1997.

[60] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Workflow evolution," in *Proceedings of the 15th International Conference on Conceptual Modeling*. Springer-Verlag, 1996, pp. 438–455.

[61] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems." Milpitas, California, United States: ACM, 1995, pp. 10–21.

[62] M. Reichert and P. Dadam, "Adept_flex - supporting dynamic changes of workflows without losing control," *J. Intell. Inf. Syst.*, vol. 10, no. 2, pp. 93–129, 1998.

[63] G. Joeris and O. Herzog, "Managing evolving workflow specifications with schema versioning and migration rules," University of Bremen, TZI Technical Report 15-1999, 1999.

[64] W. van der Aalst and S. Jablonski, "Dealing with workflow change: identification of issues and solutions," *International Journal of Computer Science*, vol. 15, no. 5, pp. 267–276, Sep. 2000.

[65] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems: a survey," *Data Knowl. Eng.*, vol. 50, no. 1, pp. 9–34, 2004.

[66] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change patterns and change support features - enhancing flexibility in process-aware information systems," *Data Knowl. Eng.*, vol. 66, no. 3, pp. 438–466, 2008.

[67] C. Peltz and A. Anagol-Subbarao, "Design strategies for web services versioning," 2004. [Online]. Available: http://soa.sys-con.com/node/44356

[68] M. Poulin, "Service versioning for SOA," 2006, published: SOAWorld Magazine vol.6(7). [Online]. Available: http://webservices.sys-con.com/read/250503.htm

[69] M. B. Juric, A. Sasa, B. Brumen, and I. Rozman, "WSDL and UDDI extensions for version support in web services," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1326–1343, Aug. 2009.

[70] M. B. Juric, A. Sasa, and I. Rozman, "WS-BPEL extensions for versioning," *Information and Software Technology*, vol. 51, no. 8, pp. 1261–1274, Aug. 2009.

[71] A. Narayan and I. Singh, "Designing and versioning compatible web services," Mar. 2007. [Online]. Available: http://www.ibm.com/developerworks/websphere/library/techarticles/0705_narayan/0705_narayan.html

[72] S. Becker, A. Borgi, I. Gordon, and S. Overhage, "Towards and engineering approach to component adaptation," in *Architecting Systems*, vol. 3938. Springer, 2006, pp. 193–215.

[73] C. Canal, P. Poizat, and G. Salaün, "Model-Based adaptation of behavioral mismatching components," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 546–563, 2008.

[74] X. Xiong and Z. Weishi, "A checking mechanism of software component adaptation," in *GCC '06: Proceedings of the Fifth International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–354.

[75] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: two faces of the same coin," in *ICCAD*, 2002, pp. 132–139.

[76] M. Dumas, M. Spork, and K. Wang, "Adapt or perish: Algebra and visual notation for service interface adaptation," in *Business Process Management, 4th International Conference Proceedings*, ser. Lecture Notes in Computer Science. Vienna, Austria: Springer, 2006, pp. 65–80.

[77] M. Hiel, H. Weigand, and W. van den Heuvel, "An adaptive Service-Oriented architecture," in *Enterprise Interoperability III*. Springer London, 2008, pp. 197–208.

[78] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "PAWS: a framework for executing adaptive Web-Service processes," *IEEE Software*, vol. 24, no. 6, pp. 39–46, 2007.

[79] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava, "Adaptation in web service composition and execution," in *International Conference on Web Services*, 2006, pp. 549–557.

[80] W. Kongdenfha, R. Saint-paul, B. Benatallah, and F. Casati, "An aspect-oriented framework for service adaptation," in *ICSOC 2006 Proceedings*. Chicago, IL, USA: Springer-Verlag, 2006, pp. 15—26.

[81] M. Colombo, E. D. Nitto, and M. Mauri, "SCENE: a service composition execution environment supporting dynamic changes disciplined through rules," in *ICSOC 2006*. Springer, 2006, pp. 191–202.

[82] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, 2007.

[83] D. Karastoyanova and F. Leymann, "BPEL'n'Aspects: adapting service orchestration logic," in *Proceedings of the 2009 IEEE International Conference on Web Services*. IEEE Computer Society, 2009, pp. 222–229.

[84] E. di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, no. 3-4, pp. 313–341, Dec. 2008.

[85] A. Brogi and R. Popescu, "Automated generation of BPEL adapters," in *ICSOC 2006*, ser. Lecture Notes in Computer Science. Springer, 2006, pp. 27–39.

[86] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *Proceedings of the 16th international conference on World Wide Web*. Banff, Alberta, Canada: ACM, 2007, pp. 993–1002.

[87] T. Senivongse, "Enabling flexible Cross-Version interoperability for distributed services," in *Proceedings of the International Symposium on Distributed Objects and Applications*. IEEE Computer Society, 1999, p. 201.

[88] S. R. Ponnekanti and A. Fox, "Interoperability among independently evolving web services," ser. Lecture Notes in Computer Science. Toronto, Canada: Springer Berlin / Heidelberg, 2004, pp. 331–351.

[89] P. Kaminski, M. Litoiu, and H. Müller, "A design technique for evolving web services." Toronto, Ontario, Canada: ACM, 2006, p. 23.

[90] D. Frank, L. Lam, L. Fong, R. Fang, and M. Khangaonkar, "Using an interface proxy to host versioned web services," in *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*. IEEE Computer Society, Jul. 2008, pp. 325–332.

[91] W. van den Heuvel, H. Weigand, and M. Hiel, "Configurable adapters: the substrate of self-adaptive web services," in *Proceedings of the ninth international conference on Electronic commerce*, Minneapolis, MN, USA, 2007, pp. 127–134.

[92] J. Kenyon, "Web service versioning and deprecation," Jan. 2003. [Online]. Available: http://soa.sys-con.com/node/39678

[93] K. Brown and M. Ellis, "Best practices for web services versioning," Jan. 2004. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-version/

[94] J. Evdemon, "Principles of service design: Service versioning," Aug. 2005. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms954726.aspx

[95] M. Russell, "Manage message contract changes with versioning," Aug. 2005. [Online]. Available: http://www.ibm.com/developerworks/web/library/wa-msgvers/index.html

[96] M. Endrei, M. Gaon, J. Graham, K. Hogg, and N. Mulholland, "Moving forward with web services backward compatibility," May 2006. [Online]. Available: http://www.ibm.com/developerworks/java/library/ws-soa-backcomp/index.html?ca=drs-

[97] G. Bechara, "Web services versioning," Apr. 2007. [Online]. Available: http://www.oracle.com/technology/pub/articles/web_services_versioning.html

[98] K. Jerijærvi and J. Dubray, "Contract versioning, compatibility and composability," Dec. 2008. [Online]. Available: http://www.infoq.com/articles/contract-versioning-comp2

[99] D. Parachuri and S. Mallick, "Service versioning in SOA," Dec. 2008. [Online]. Available: http://www.infosys.com/offerings/IT-services/soa-services/white-papers/pages/index.aspx

[100] G. Flurry, "Service versioning in SOA," Oct. 2008. [Online]. Available: http://www.ibm.com/developerworks/websphere/techjournal/0810_col_flurry/0810_col_flurry.html

[101] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen, and N. Du, "A version-aware approach for web service directory," in *ICWS 2007*, Jul. 2007, pp. 406–413.

[102] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-End versioning support for web services," in *IEEE International Conference on Services Computing, 2008.*, vol. 1, Jul. 2008, pp. 59–66.

[103] R. Weinreich, T. Ziebermayr, and D. Draheim, "A versioning model for enterprise services," in *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*, vol. 2, 2007, pp. 570–575.

[104] K. Becker, A. Lopes, D. S. Milojicic, J. Pruyne, and S. Singhal, "Automatically determining compatibility of evolving services," in *ICWS 2008*, 2008, pp. 161–168.

[105] D. Orchard, "Extending and versioning languages: Terminology [Editorial draft]," World Wide Web Consortium (W3C), Nov. 2007. [Online]. Available: http://www.w3.org/2001/tag/doc/versioning

[106] D. Orchard *Ed.*, "Extending and versioning languages: XML languages [Editorial draft]," World Wide Web Consortium (W3C), Jul. 2007. [Online]. Available: http://www.w3.org/2001/tag/doc/versioning-xml

[107] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou, "On the evolution of services," *IEEE Transactions on Software Engineering (under revision)*, 2010.

[108] M. Treiber, H. Truong, and S. Dustdar, "On analyzing evolutionary changes of web services," in *ICSOC 2008 International Workshops, Revised Selected Papers*, ser. Lecture Notes in Computer Science. Sydney, Australia: Springer-Verlag, 2008, pp. 284–297.

[109] M. Treiber, H.-L. Truong, and S. Dustdar, "SEMF - service evolution management framework," in *Software Engineering and Advanced Applications, 2008. SEAA '08*, 2008, pp. 329–336.

[110] M. Kajko-Mattsson, G. A. Lewis, and D. B. Smith, "Evolution and maintenance of SOA-Based systems at SAS," in *Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, 2008, p. 119.

[111] L. Pasquale, J. Laredo, H. Ludwig, K. Bhattacharya, and B. Wassermann, "Distributed Cross-Domain configuration management," in *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*. Stockholm: Springer-Verlag, 2009, pp. 622–636.

[112] B. Wassermann, H. Ludwig, J. Laredo, K. Bhattacharya, and L. Pasquale, "Distributed Cross-Domain change management," in *Proceedings of the 2009 IEEE International Conference on Web Services*. IEEE Computer Society, 2009, pp. 59–66.

[113] O. von Susani and P. Dugerdil, "Cross-organizational service maintenance using temporal availability specification and contracts," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005, pp. 230–239.

[114] O. von Susani and P. Dugerdil, "Cross-Organizational service evolution management," in *Third International Conference on Information Technology: New Generations*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 332–337.

[115] M. Treiber, V. Andrikopoulos, and S. Dustdar, "Calculating service fitness in service networks," in *2nd Workshop on Monitoring, Adaptation and Beyond (MONA+)*, Stockholm, Sweden, Nov. 2009, p. (with publisher).

[116] M. Bitsaki, O. Danylevych, W. van den Heuvel, G. Koutras, F. Leymann, M. Mancioppi, C. Nikolaou, and M. Papazoglou, "An architecture for managing the lifecycle of business goals for partners in a service network," in *Proceedings of the 1st European Conference on Towards a Service-Based Internet*. Madrid, Spain: Springer-Verlag, 2008, pp. 196–207.

[117] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[118] A. Alves *et al. Eds.*, "Web services business process execution language," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[119] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and Ümit Yalçinalp *Eds.*, "Web services policy (WS-Policy) 1.5 - framework," Sep. 2007. [Online]. Available: http://www.w3.org/TR/ws-policy/

[120] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma, "Management applications of the web service offerings language (WSOL)," *Information Systems*, vol. 30, no. 7, pp. 564–586, Nov. 2005.

[121] J. Walkerdine, J. Hutchinson, P. Sawyer, G. Dobson, and V. Onditi, "A faceted approach to service specification," in *Proceedings of the Second International Conference on Internet and Web Applications and Services*. IEEE Computer Society, 2007, p. 20.

[122] J. A. Estefan, K. Laskey, F. G. McCabe, and D. Thornton *Eds.*, "OASIS service oriented architecture reference architecture version 1.0," 2008. [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html

[123] CBDI-SAE, "CBDI-SAE^{TM}meta model for SOA version 2.0," Everware-CBDI Inc, 2007. [Online]. Available: http://www.cbdiforum.com/public/meta_model_v2.php

[124] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, Jul. 2004.

[125] B. Meyer, *Object-Oriented Software Construction (2nd ed.)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[126] P. Helland, "Data on the outside versus data on the inside," in *Second Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, 2005, pp. 144–153. [Online]. Available: http://www.cidrdb.org/cidr2005/papers/P12.pdf

[127] T. Erl, *SOA Design Patterns*, 1st ed. Prentice Hall PTR, Jan. 2009.

[128] A. Keller, G. Kar, H. Ludwig, A. Dan, and J. Hellerstein, "Managing dynamic services: a contract based approach to a conceptual architecture," in *IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy, Apr. 2002, pp. 513–528.

[129] F. Buccafurri, P. D. Meo, M. Fugini, R. Furnari, A. Goy, G. Lax, P. Lops, S. Modafferi, B. Pernici, D. Redavid, G. Semeraro, and D. Ursino, "Analysis of QoS in cooperative services for real time applications," *Data & Knowledge Engineering*, vol. 67, no. 3, pp. 463–484, Dec. 2008.

[130] M. Comuzzi and B. Pernici, "A framework for QoS-based web service contracting," *ACM Transactions on the Web*, vol. 3, no. 3, pp. 1–52, 2009.

[131] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou, "Evolving services from a contractual perspective," in *Proceedings of the 21st international conference on Advanced Information Systems Engineering*. Amsterdam, the Netherlands: Springer-Verlag, 2009, pp. 290–304.

[132] Y. Zhong and J. Yang, "Contract-First design techniques for building enterprise web services," in *Proceedings of the 2009 IEEE International Conference on Web Services*. IEEE Computer Society, 2009, pp. 591–598.

[133] R. Kazhamiakin *Ed.*, "CD-IA-3.2.1 initial definition of validation scenarios," S-Cube Consortium, Oct. 2009. [Online]. Available: http://www.s-cube-network.eu/

[134] A. Gehlert and A. Metzger *Eds.*, "CD-JRA-1.3.2 quality reference model for SBA," S-Cube Consortium, Mar. 2008. [Online]. Available: http://www.s-cube-network.eu/

[135] A. Nadalin, R. Monzillo, and P. Hallam-Baker *Eds.*, "WS-Security - SOAP message security 1.1," Feb. 2006. [Online]. Available: http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[136] B. Benatallah, F. Casati, and F. Toumani, "Representing, analysing and managing web service protocols," pp. 327–357, 2006.

[137] S. H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul, "Supporting the dynamic evolution of web service protocols in service-oriented architectures," *ACM Transactions on the Web*, vol. 2, no. 2, pp. 1–46, 2008.

[138] M. Mancioppi, M. Carro, W. van den Heuvel, and M. P. Papazoglou, "Sound multi-party business protocols for service networks," in *Proceedings of the 6th International Conference on Service-Oriented Computing*. Sydney, Australia: Springer-Verlag, 2008, pp. 302–316.

[139] J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt, "Formal specification of a web services protocol," *Electronic Notes in Theoretical Computer Science*, vol. 105, pp. 147–158, Dec. 2004.

[140] D. Beyer, A. Chakrabarti, and T. A. Henzinger, "Web service interfaces," in *Proceedings of the 14th international conference on World Wide Web*. Chiba, Japan: ACM, 2005, pp. 148–159.

[141] E. Elabd, E. Coquery, and M. Hacid, "Compatibility and replaceability analysis of timed web services protocols," in *Computer and Electrical Engineering, International Conference on*, vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 15–19.

[142] J. Ponge, B. Benatallah, F. Casati, and F. Toumani, "Analysis and applications of timed service protocols," *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 4, pp. 1–38, 2010.

[143] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani, "A formal account of contracts for web services," in *Web Services and Formal Methods - Third International Workshop Proceedings*. Springer, 2006, pp. 148–162.

[144] G. Castagna, N. Gesbert, and L. Padovani, "A theory of contracts for web services," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 5, pp. 1–61, 2009.

[145] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, 1994.

[146] C. Rolland, M. Kirsch-Pinheiro, and C. Souveyet, "An intentional approach to service engineering," *IEEE Transactions on Services Computing (to appear)*, 2010.

[147] A. D. Ambrogio, "A model-driven WSDL extension for describing the QoS of web services," in *Web Services, IEEE International Conference on*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 789–796.

[148] D. A. Menascé, "QoS issues in web services," *IEEE Internet Computing*, vol. 6, no. 6, pp. 72–75, 2002.

[149] J. O'Sullivan, D. Edmond, and A. T. Hofstede, "What's in a service?" *Distrib. Parallel Databases*, vol. 12, no. 2-3, pp. 117–133, 2002.

[150] M. Tian, A. Gramm, H. Ritter, J. Schiller, and R. Winter, "A survey of current approaches towards specification and management of quality of service for web services," *Praxis der Informationsverarbeitung und Kommunikation*, vol. 27, no. 3, pp. 132–139, 2004.

[151] S. Benbernou, I. Brandic, C. Cappiello, M. Carro, M. Comuzzi, A. Kertész, K. Kritikos, M. Parkin, B. Pernici, and P. Plebani, "A survey on service quality description," *ACM Computing Surveys (under review)*, 2010.

[152] K. Mokhtari, S. Benbernou, M. Rouached, Mohand-Said Hacid, and F. Leymann, "Privacy Time-Related analysis in business protocols," in *IEEE International Conference on Web Services*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 141–148.

[153] V. Andrikopoulos, M. Fugini, M. P. Papazoglou, M. Parkin, B. Pernici, and H. Siadat, "QoS contracts formation & evolution," in *11th International Conference on Electronic Commerce and Web Technologies (to appear)*, 2010.

[154] A. Erradi, S. Padmanabhuni, and N. Varadharajan, "Differential QoS support in web services management." IEEE Computer Society, 2006, pp. 781–788.

[155] L. Cardelli, "A semantics of multiple inheritance," *Inf. Comput.*, vol. 76, no. 2-3, pp. 138–164, 1988.

[156] S. Vinoski, "The more things change..." *IEEE Internet Computing*, vol. 8, no. 1, pp. 87–89, 2004.

[157] D. Booth and C. K. Liu *Eds.*, "Web services description language (WSDL) version 2.0 part 0: Primer," http://www.w3.org/TR/wsdl20-primer, 2007. [Online]. Available: http://www.w3.org/TR/wsdl20-primer

[158] L. Clement, A. Hately, C. von Riegen, and T. Rogers *Eds.*, "UDDI version 3.0.2," 2004. [Online]. Available: http://www.uddi.org/pubs/uddi_v3.htm

[159] M. Belguidoum and F. Dagnat, "Formalization of component substitutability," *Electron. Notes Theor. Comput. Sci.*, vol. 215, pp. 75–92, 2008.

[160] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–523, 1985.

[161] D. Orchard, "Extensibility, XML vocabularies, and XML schema," Oct. 2004. [Online]. Available: http://www.xml.com/lpt/a/1492

[162] P. America, "Designing an Object-Oriented programming language with behavioural subtyping," in *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*. Springer-Verlag, 1991, pp. 60–90.

[163] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[164] Y. Liu, S. Müller, and K. Xu, "A static compliance-checking framework for business process models," *IBM Systems Journal*, vol. 46, no. 2, pp. 335–361, 2007.

[165] Y. Velegrakis, R. J. Miller, and L. Popa, "Preserving mapping consistency under schema changes," *The VLDB Journal*, vol. 13, no. 3, pp. 274–293, 2004.

[166] A. Fuxman, M. A. Hernandez, H. Ho, R. J. Miller, P. Papotti, and L. Popa, "Nested mappings: schema mapping reloaded," in *Proceedings of the 32nd international conference on Very large data bases*. Seoul, Korea: VLDB Endowment, 2006, pp. 67–78.

[167] I. Brandic, S. Venugopal, M. Mattess, and R. Buyya, "Towards a Meta-Negotiation architecture for SLA-Aware grid services," in *Workshop on Service-Oriented Engineering and Optimizations 2008*, Bangalore, India, 2008.

[168] J. Vara, D. Granada, V. Andrikopoulos, and E. Marcos, "Modeling and comparing service descriptions," University Rey Juan Carlos, Department of Computing Languages and Systems II, Madrid, Spain, Technical Report TR-29032010, Jun. 2010. [Online]. Available: http://kybele.es/research/TR/TR-29032010.pdf

[169] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Addison-Wesley Professional, Aug. 2003.

[170] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2009-28, 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[171] L. Richardson, S. Ruby, and D. H. Hanson, *RESTful web services*. O'Reilly Media, 2007.

[172] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, chair-Taylor, Richard N.

[173] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful web services vs. "big"' web services: making the right architectural decision," in *Proceeding of the 17th international conference on World Wide Web*. Beijing, China: ACM, 2008, pp. 805–814.

[174] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary Web service discovery mechanisms," *Journal of Web Engineering*, vol. 5, no. 3, pp. 265–290, 2006.

[175] P. Plebani and B. Pernici, "URBE: Web service retrieval based on similarity evaluation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 11, pp. 1629–1642, 2009.

[176] M. Akram, B. Medjahed, and A. Bouguettaya, "Supporting dynamic changes in web service environments," in *Service-Oriented Computing*, 2003, pp. 319–334.

[177] X. Liu and A. Bouguettaya, "Managing top-down changes in Service-Oriented enterprises," in *Web Services, IEEE International Conference on.* Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1072–1079.

Author Index

Ahmed-Nacer 24
Akram 161
Allen 94
Alves 33, 143
Ambrogio 56
America 88
Anagol-Subbarao 25, 71, 72, 75
Andany 24
Andrikopoulos 7, 30, 32, 36, 51, 54, 56, 57, 59, 64, 84, 88, 94, 106, 110, 111, 132
Antón 19
Ardagna 27
Armbrust 160
Arnold 19

Banerjee 23, 64
Basili 22
Bechara 29, 72
Becker 26, 30, 87
Belguidoum 82
Benatallah 24, 27, 28, 54
Benbernou 7, 30, 36, 51, 54, 56, 57, 59, 64, 84, 106, 110, 111
Bennett 19, 20
Bertino 24
Beugnard 32, 35
Beyer 54
Bhattacharya 31, 106
Bitsaki 32, 161
Boehm 22
Bohner 19
Booch 35, 40, 52
Booth 73, 83, 143
Borgi 26
Boubez 33, 56, 143
Bouguettaya 6, 24, 161
Brada 23, 106
Brandic 56, 128
Bretl 24
Brogi 28, 54
Brooks 3
Brown 22, 29, 71, 72, 86
Brumen 25, 72, 73, 75
Buccafurri 36, 56
Buckley 17
Budinsky 132
Buyya 128

Caires 23, 106
Canal 26
Canfora 17, 22, 28
Cappiello 56
Cardelli 57, 84, 87, 88
Carpineti 54, 91
Carro 54, 56
Casati 24, 27, 28, 54
Castagna 54, 55, 88, 91, 110
CBDI-SAE 34
Ceri 24
Chafle 27
Chakrabarti 54
Chapin 17, 18
Chen 29, 72, 75
Chou 24
Christianson 5, 19
Clark 22

Clement 73
Clemm 20, 68, 69
Colombo 27
Comuzzi 27, 36, 56, 111
Conradi 20, 68, 69, 71
Coquery 54
Curbera 28, 54

Dadam 25
Dagnat 82
Dan 36
Danylevych 32, 161
Dasgupta 27
de Alfaro 26
di Nitto 28
Dobson 34, 55
Draheim 30, 72, 76
Du 29, 72, 75
Dubray 29, 72, 85, 86
Dugerdil 32, 128
Dumas 27, 28, 54
Dustdar 5, 29, 31–33, 71–73, 159

Edmond 24, 56
Elabd 54
Elfatatry 22
Ellersick 132
Ellis 24, 29, 71, 72, 86
Endrei 29, 72, 75, 86
Erl 36
Erradi 57
Esfandiari 34, 56
Estefan 34, 144
Estublier 20, 24, 68, 69
Evdemon 29, 35, 72, 85

Fang 28, 29, 72, 75
Fielding 160
Flurry 29, 72
Fong 28, 29, 72, 75
Fowler 4
Fox 28, 29, 160
Frank 28, 29, 72, 75
Fugini 36, 56, 88, 94, 110
Furnari 36, 56
Fuxman 119

Gaon 29, 72, 75, 86
Garofalakis 160
Gehlert 43, 56
Georgakopoulos 4
Gesbert 54, 55, 88, 91, 110
Ghezzi 28
Gordon 26
Goy 36, 56
Graham 29, 72, 75, 86
Gramm 56
Granada 132
Grandi 24
Griffith 160
Grose 132
Guerrini 24

Hacid 54
Hale 17, 18
Hallam-Baker 44, 56
Hanson 160
Hately 73
Helland 35
Hellerstein 36
Henzinger 26, 54
Hernandez 119
Herzog 25
Hevner 11, 156
Hewitt 5, 19
Hiel 27, 28
Hirsch 33, 56, 143
Ho 119
Hofstede 56
Hogg 29, 72, 75, 86
Hondo 33, 56, 143
Hutchinson 34, 55

Jablonski 25
Jacobson 35, 40, 52
Jerijærvi 29, 72, 85, 86
Jézéquel 32, 35
Joeris 25

Johnson 54
Joseph 160
Juric 25, 72, 73, 75

Kajko-Mattsson 31
Kaminski 28, 29
Kar 36
Karastoyanova 27
Katz 160
Kazhamiakin 39, 40
Keddara 24
Keller 36
Kenyon 29, 72
Kertész 56
Khan 17, 18
Khangaonkar 28
Kim 23, 24, 64
Kirsch-Pinheiro 55, 56
Kniesel 17
Kongdenfha 27
Konwinski 160
Korth 23, 64
Koutras 32, 161
Krämer 24
Kritikos 56
Kumar 27

Lam 28, 29, 72, 75
Lamport 54
Laneve 54, 91
Langworthy 54
Laredo 31, 106
Laskey 34, 144
Lax 36, 56
Leblang 20, 68, 69
Lee 160
Lehman 18, 22
Leitner 29, 71–73, 159
Léonard 24
Lewis 31
Leymann 5, 27, 32, 33, 56, 160, 161
Lientz 17
Liskov 55, 88
Litoiu 28, 29
Liu 6, 106, 161
Liu 73, 83, 143
Lopes 30, 87
Lops 36, 56
Ludwig 31, 36, 106

Ma 34, 56
Maier 24
Mallick 29, 72, 76, 85, 86
Mancioppi 32, 54, 161
Mandreoli 24
March 11, 156
Marcos 132
Martens 28, 54
Martino 24
Mattess 128
Mauri 27
McCabe 34, 144
Medjahed 6, 161
Menascé 56
Mens 17, 19
Meo 36, 56
Merks 132
Metzger 28, 43, 56
Meyer 35, 55, 88
Michlmayr 29, 71–73, 159
Miller 119
Milojicic 30, 87
Mittal 27
Mittermeir 19
Modafferi 36, 56
Mohand-Said Hacid 56
Mokhtari 56
Monk 24
Monzillo 44, 56
Mulholland 29, 72, 75, 86
Müller 28, 29, 106
Mussi 27

Nadalin 44, 56
Narayan 25, 29, 72, 75, 85
Narayanaswamy 20

Nehaniv 5, 19
Nezhad 28, 54
Nikolaou 32, 161
Nitto 27

Onditi 34, 55
Orchard 6, 30, 33, 56, 85, 143
Orchard 30, 82, 84, 85, 143
O'Sullivan 56
Otis 24
Overhage 26
Özsu 24

Padmanabhuni 57
Padovani 54, 55, 88, 91, 110
Pagurek 34, 56
Palisser 24
Panagis 160
Papazoglou 4, 5, 7, 8, 22, 24, 28, 30, 32, 33, 36, 51, 54, 56, 57, 59, 64, 70, 84, 88, 94, 106, 110, 111, 161
Papotti 119
Parachuri 29, 72, 76, 85, 86
Park 11, 156
Parkin 56, 88, 94, 110
Parnas 4, 6
Pasquale 31, 106
Passerone 26
Patel 34, 56
Patterson 160
Pautasso 160
Peltz 25, 71, 72, 75
Penney 24
Pernici 24, 27, 36, 56, 88, 94, 110, 111, 160
Peters 24
Plebani 27, 56, 160
Plouzeau 32, 35
Pohl 28
Poizat 26
Ponge 54
Ponnekanti 28, 29
Popa 119
Popescu 28, 54
Potts 19
Poulin 25, 72, 75
Pozzi 24
Pruyne 30, 87

Rabkin 160
Rajlich 19, 20
Ram 11, 156
Ramil 17, 18, 22
Rashid 17
Redavid 36, 56
Reichert 25
Reifer 22
Richardson 160
Rinderle 25
Rinderle-Ma 25
Ritter 56
Robinson 5, 36, 128, 141, 142
Rogers 73
Rolland 55, 56
Rosenberg 29, 71–73, 159
Rouached 56
Rozenberg 24
Rozman 25, 72, 73, 75
Ruby 160
Rumbaugh 35, 40, 52
Rusca 24
Russell 29, 72
Ryu 54

Saint-Paul 54
Sakkopoulos 160
Salaün 26
Sangiovanni-Vincentelli 26
Sasa 25, 72, 73, 75
Sawyer 34, 55
Scacchi 20
Schiller 56
Schuchardt 24
Seco 23, 106
Semeraro 36, 56
Senivongse 28, 29
Siadat 56, 88, 94, 110

Singh 25, 29, 72, 75, 85
Singhal 30, 87
Skogsrud 54
Smith 11, 31
Sommerville 24
Souveyet 55, 56
Sowrirajan 21
Spork 27, 28, 54
Sprott 5
Srivastava 27
Stein 24
Steinberg 132
Stoica 160
Stuckenholz 22
Swanson 17
Szyperski 22

Tan 17, 18
Thornton 34, 144
Tian 56
Tichy 20, 68, 69
Tosic 34, 56
Toumani 54
Tourwé 19
Traverso 5, 33
Treiber 31, 32
Truong 31
Tsakalidis 160

Yalçinalp 33, 56, 143
Ursino 36, 56

van den Heuvel 4, 27, 28, 32, 54, 161
van der Aalst 25
van der Hoek 20, 21, 68, 69
Vara 132
Varadharajan 57
Vedamuthu 33, 56, 143
Velegrakis 119
Venugopal 128
Vignola 29, 72, 75
Vinoski 70
Vogt 54
von Riegen 73
von Susani 32, 128

Walkerdine 34, 55
Wallnau 22
Wang 27, 28, 54
Wassermann 31, 106
Watkins 32, 35
Weber 25
Wegner 84, 87
Weigand 27, 28
Weinreich 30, 72, 76
Weishi 26
Wernick 5, 19
Westfechtel 20, 68, 71
Wiborg-Weber 20, 68, 69
Williams 24
Wing 55, 88
Winter 56

Xiong 26
Xu 106

Yang 36, 110
Yendluri 33, 56, 143
Yu 6, 161

Zaharia 160
Zdonik 24
Zenger 17, 23, 106
Zhong 36, 110
Zicari 23
Ziebermayr 30, 72, 76
Zimmermann 160

Index

ϑ function, 117, 119, 120, 126
PopClient, 109
PopService, 38

Abstract Service Description, *see* ASD
adaptation
  component, 24
  self-, 26
  service, 3, 24, 26
    composition, 25
    interfaces, 25
adapter, 26
  chain of, 26
  component, 24
Allen's Interval Algebra, 92
AOP, 25
ASD, 50, 61
  $\mathcal{S}^{xpe}$, 114
  $\mathcal{S}^{xpo}$, 114
  $\mathcal{S}_{net}$, 114
  $\mathcal{S}_{pro}$, 82, 113
  $\mathcal{S}_{req}$, 82, 113
  element, 50, 56
    `Activity`, 59
    `Assertion`, 60, 92
    `Protocol`, 59, 89
  layers, 50
  Meta-model, 51
  record, 50
    versioned, 76
  relationship, 50, 56
    `lType`, 60, 95
  versioned, 76
  views, 113
ASD concepts
  behavioral, 53
  mappings to WS-*, 56
  non-functional, 54
  structural, 50
Aspect-Oriented Programming, *see* AOP

behavioral specification, 53
behavioral subcontracting, 89
binding function, *see* ϑ function
BPEL, 6, 31, 53, 141
business process, 3
Business Process Execution Language, *see* BPEL
business protocol, 6, 52

CBS, 20, 30
change patterns, 27, 29, 101
  in workflows, 23
change primitives, 77
change scenarios, 42, 98, 112, 139
change set, 77, 96
class invariance, 33
closure, 156
Commercial-Off-The-Shelf systems, *see* COTS
compatibility
  backward, 27, 73, 80, 82–84, 141
    guidelines for, 84, 101
  check, 97, 133, 142
  component, 80
  contract, 124
  contract-based, 123
  forward, 80, 82, 83, 141

full, 82
guidelines for, 27
horizontal, 80, 107
service, 7, 11, 27, 80
vertical, 80
Compatibility Checking Algorithm, 97
completeness, 156
component, 20
Component-Based Systems, *see* CBS
consistency
ASD, 62, 132
schema, 21
workflow, 23
context, 80
contract
behavioral, 52, 59, 89
broker, 108
clauses, 119
compliance to, 122
configuration, 109, 119
policies, 120
consumer-driven, 34
legal, 33
service, 11, 30, 33, 108, 119
life cycle, 108
software, 33
Contract Formation Algorithm, 121
contravariance, 82
correctness
schema, 22
workflow, 23
COTS, 20
covariance, 82

deep changes, 5, 101, 104, 159
delta, 66, 77
design science, 9

Eclipse platform, 130
Eiffel programming language, 33
encapsulation, 2, 6, 8
evolution
compatible
service, 83
compatible service, 23
component, 20
distributed computing, 17
in biology, 2, 17
language, 4, 28, 82
object, 21
service, 2
approaches to, 23
strategies for, 28
software, 15
laws of, 16
workflow, 22
expectation/exposition view, 114
extensibility, 83, 142

functional paleontology, 17

IDL, 30
Interface Definition Language, *see* IDL
interoperability, 80, 107
invariants, 21, 62
inversion operator, 116

language producers & consumers, 4, 80
loosely coupled, 8

marshaling, 142
matchmaking, 109, 117
message exchange pattern, 81

object evolution, *see* evolution, object
object-oriented databases, 21
obligation property, 54, 94, 111
OWL-S, 32

Postel's Law, 150
product space, 66
promise property, 54, 93
provided/required view, 113
provider selection, 109

QoS, 7
characteristics, 53, 60
dimension, 54

antitonic, 54, 92
monotonic, 54, 92
ordinal, 54
QRM, 41, 54
Quality of Service, *see* QoS
Quality Reference Model, *see* QRM

replaceability, 80
RESTful services, 29, 158
reusability, 2
revision control systems, 19

SBA, 4, 29, 101, 109, 114, 137
schema matching, 117
SCM, 18
for SBAs, 29
SCOR, 37
service adaptation, *see* adaptation, service
service binding, 141
service chain, 38
service change management, 29
service composition, 3, 6, 31
service decommission, 30
service description, 6, 30, 141
service evolution, *see* evolution, service
service facets, 32
service fitness, 30
service implementation, 6
service interface, 6, 50
Service Level Agreement, *see* SLA
service life cycle, 4, 26, 68
change-oriented, 6, 159
service mapping, 119
service matching, 117
service network, 30
Service Oriented Architecture, *see* SOA
service proxy, 26, 27
service registry, 71
service representation, 11, 33, 49
Service Representation Modeler, *see* SRM
service signatures, 6
service stakeholders, 29
service version, 3, 5
Service-Based Application, *see* SBA
set theory, 28
shallow changes, 5, 99, 126
condition for, 82
SLA, 34, 108, 126
SOA, 2, 3
Meta Model for, 32, 49
Reference Architecture, 32, 49
software
component, *see* component
evolution, *see* evolution, software
versioning, *see* Versioning, software
software aging, 4
Software Configuration Management, *see* SCM
software maintenance, 15
SRM, 130
substitutability, 80
subtyping, 86
`Assertion Set`, 95
`Assertion`, 92
`Profile`, 95
`Protocol`, 89
behavioral, 86, 89
elements, 87
non-functional, 91
record, 82
relationships, 87
structural, 87
Supply Chain Operations Reference, *see* SCOR
supply chains, 37
system desing, 1

T-shaped changes, 29, 80, 96, 126
type system, 21, 141
type theory, 55, 82, 85
for components, 21
for services, 86

UDDI `tModel`, 72
UML relationships semantics, 57

version identifier, *see* VID

version model, 66
version space, 66
versioning, 65
    change set, 67
    component, 20
    decommissioning strategies, 73
    extensional, 66
    in object-oriented databases, 22
    in software project management, 22
    in temporal databases, 22
    intensional, 66
    service, 11, 19, 23, 27
    software, 18, 66
VID, 66, 71
    for ASDs, 76
    naming, 69

Web Services Description Language, *see* WSDL
WS-Policy, 31, 54, 141
WSDL, 6, 31, 51, 141
WSOL, 31

XML namespace, 71, 142
XML Schema, 50
XML wildcards, 83

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/ Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennis-technologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

Modeling Techniques

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)

A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlović (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA)
Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development and management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular,continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UVA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying

Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)
Understanding Classification

2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA)
Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA)
Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
Operating Guidelines for Services

2009-13 Steven de Jong (UM)
Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU)
From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA)
Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT)
New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT)
Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI)
Armada, An Evolving Database System

2009-19 Valentin Robu (CWI)
Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU)
Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM)
Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT)
Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU)
Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA)
Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI)
RAM: Array Database Management through Relational Mapping

2009-26 Fernando Koch (UU)
An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU)
Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT)
Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT)
Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI)
Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA)
A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU)
Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT)
How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU)
Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL)
Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN)
Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN)
Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU)
Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin)
Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT)
Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT)
Digital Analysis of Paintings

2009-42 Toine Bogers
Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT)
Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT)
Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU)
Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA)
Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU)
Patterns that Matter

2010-02 Ingo Wassink (UT)
Work flows in Life Science

2010-03 Joost Geurts (CWI)
A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT)
Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT)
Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT)
Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT)
Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL)
Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL)
A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL)
Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD)
The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU)
Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN)
High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU)
Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT)
Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD)
Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU)
Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU)
Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA)
People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT)
Whose Story Is It Anyway? How Improv Informs

Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT)
Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI)
End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU)
The Logical Structure of Emotions

2010-24 Dmytro Tykhonov
Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU)
Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI)
XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL)
Automatisch contracteren

2010-28 Arne Koopman (UU)
Characteristic Relational Patterns

2010-29 Stratos Idreos (CWI)
Database Cracking: Towards Auto-tuning Database Kernels

2010-30 Marieke van Erp (UvT)
Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval

2010-31 Victor de Boer (UVA)
Ontology Enrichment from Heterogeneous Sources on the Web

2010-32 Marcel Hiel (UvT)
An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems

2010-33 Robin Aly (UT)
Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT)
Interaction Design in Service Compositions

2010-35 Dolf Trieschnigg (UT)
Proof of Concept: Concept-based Biomedical Information Retrieval

2010-36 Jose Janssen (OU)
Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification

2010-37 Niels Lohmann (TUE)
Correctness of services and their composition

2010-38 Dirk Fahland (TUE)
From Scenarios to components

2010-39 Ghazanfar Farooq Siddiqui (VU)
Integrative modeling of emotions in virtual agents

2010-40 Mark van Assem (VU)
Converting and Integrating Vocabularies for the Semantic Web

2010-41 Guillaume Chaslot (UM)
Monte-Carlo Tree Search

2010-42 Sybren de Kinderen (VU)
Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach

2010-43 Peter van Kranenburg (UU)
A Computational Approach to Content-Based Retrieval of Folk Song Melodies

2010-44 Pieter Bellekens (TUE)
An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain

2010-45 Vasilios Andrikopoulos (UvT)
A theory and model for the evolution of software services